Verification-Aware Processor Design

by

Anita Lungu

Department of Computer Science
Duke University

Date:_______________________

Approved:

___________________________

Professor Daniel J. Sorin, Advisor

___________________________

Professor Landon Cox

___________________________

Professor Christopher Dwyer

___________________________

Professor Alvin R. Lebeck

___________________________

Professor Jun Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2009
ABSTRACT

Verification-Aware Processor Design

by

Anita Lungu

Department of Computer Science
Duke University

Date:_______________________

Approved:

___________________________
Professor Daniel J. Sorin, Advisor

___________________________
Professor Landon Cox

___________________________
Professor Christopher Dwyer

___________________________
Professor Alvin R. Lebeck

___________________________
Professor Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2009
Copyright © 2009
by
Anita Lungu
All rights reserved
Abstract

As technological advances enable computers to permeate many of our society’s critical application domains (such as medicine, finances, transportation), the requirement for computers to always behave correctly becomes critical as well. Currently, ensuring that processor designs are correct represents a major challenge for the computing industry consuming the majority (up to 70%) of the resources allocated for the creation of a new processor. Looking towards the future, we see that with each new processor generation, even more transistors fit on the same chip area and more complex designs become possible, which makes it unlikely that the difficulty of the design verification problem will decrease by itself.

We believe that the difficulty of the design verification problem is compounded by the current processor design flow. In most design cycles, a design’s verifiability is not explicitly considered at an early stage – when decisions are most influential – because that initial focus is exclusively on improving the design on more traditional metrics like performance, power, and area. It is thus possible for the resulting design to be very difficult to verify in the end, specifically because its verifiability was not ranked high on the priority list in the beginning.

In this thesis we propose to view verifiability as a critical design constraint to be considered, together with other established metrics, like performance and power, from the initial stages of design. Our high level goal is for this approach to make designs
more verifiable, which would both decrease the resources invested in the verification step and lead to more robust designs.

More specifically, we make five main contributions in this thesis. The first is our proposal for a change in design perspective towards considering verifiability as a first class constraint. Second, we use formal verification (through a combination of theorem proving, model checking, and probabilistic model checking) to quantitatively evaluate the impact on verifiability of various design choices like the organization of caches, TLBs, pipeline, operand bypass network, and dynamic power management mechanisms. Our third contribution is to evaluate design trade-offs between verifiability and other established metrics, like performance and power, in the context of multi-core dynamic power management schemes. Fourth, we re-design several components for increasing their verifiability. Finally, we propose design guidelines for increasing verifiability. In the context of single core processors our guidelines refer to the organization of caches and translation lookaside buffers (TLBs), the depth of the core’s pipeline, the type of ALUs used, while for multi-core processors we refer to dynamic power management schemes (DPMs) for power capping.

Our results confirm that making design choices with verifiability as a first class design constraint has the capacity to decrease the verification effort. Furthermore, making explicit trade-offs between verifiability, performance and power helps identify better design points for given verification, performance, and power goals.
Dedication

To my parents: Eugenia and Marin.

Parintilor mei: Eugenia si Marin.
Contents

Abstract .............................................................................................................................................iv

List of Tables ....................................................................................................................................xii

List of Figures .................................................................................................................................. xiii

Acknowledgements ........................................................................................................................xv

Introduction ......................................................................................................................................1

1.1 Research related to verification-aware design ........................................................................7

1.1.1 Design for verifiability ........................................................................................................7

1.1.2 Complexity-effective design ...............................................................................................9

1.1.3 Improving individual steps in the verification process ....................................................10

1.2 Thesis Statement and Contributions ...................................................................................11

1.3 Thesis Outline ........................................................................................................................13

2. Background in Design Verification .........................................................................................14

2.1 Design verification definition and approaches ....................................................................14

2.2 Formal verification through model checking .........................................................................17

2.2.1 Non-probabilistic versus probabilistic model checking ................................................19

2.2.2 Explicit state versus symbolic model checking ...............................................................21

2.2.3 Complete versus bounded model checking .....................................................................23

2.3 Formal verification through theorem proving .......................................................................24

2.4 Combining model checking and theorem proving ...............................................................25

2.4.1 Cone of influence reduction ..............................................................................................26

2.4.2 Refinement maps ...............................................................................................................27
3.4.2 Dynamic verification ............................................................................................54
3.5 Conclusion ..................................................................................................................55
4. Verification-aware design for multi-core power management schemes ..........57
  4.1 Formal verification methodology ............................................................................58
    4.1.1 Objective of using formal verification ............................................................59
    4.1.2 Formal verification tool ........................................................................................62
  4.2 DPM Design Space Exploration ..............................................................................64
    4.2.1 High Level View of DPM Design Space ............................................................64
    4.2.2 DPM design goals and parameters .................................................................66
    4.2.3 Further motivating DPM early formal analysis .................................................67
    4.2.4 Specification of correctness ..............................................................................68
    4.2.5 DPM model ............................................................................................................69
    4.2.6 Verification process ...............................................................................................74
  4.3 Case studies in verification of multi-core power management schemes ..........74
    4.3.1 Scope of analysis .................................................................................................74
    4.3.2 Impact of number of voltage levels .................................................................76
    4.3.3 Impact of number of frequency levels ...............................................................78
    4.3.4 Using a homogeneous versus a heterogeneous scheme ..................................80
  4.4 Guidelines for verifiability .......................................................................................81
  4.5 Related work .............................................................................................................82
    4.5.1 Multicore power management .............................................................................82
    4.5.2 Verifying power management schemes ............................................................83
5.6.1 Simulation methodology ................................................................. 112
5.6.2 Energy Model .................................................................................. 115
5.7 Experimental results .......................................................................... 117
5.7.1 Worst-case energy penalty ............................................................... 118
5.7.2 Average-case energy savings .......................................................... 119
5.7.3 Performance .................................................................................... 121
5.8 Related work ..................................................................................... 122
5.8.1 Power gating mechanisms and algorithms ..................................... 123
5.8.2 Energy management ....................................................................... 124
5.8.3 Speculation control ........................................................................ 124
5.8.4 Other mechanisms for reducing leakage ........................................ 125
5.9 Conclusions ....................................................................................... 125
6. Summary and Conclusions .................................................................. 127
References ............................................................................................... 131
Biography ................................................................................................. 139
List of Tables

Table 3-1: Properties to verify. .................................................................40
Table 3-2: Verification effort of properties as function of cache and TLB associativity...45
Table 3-3: Verifiability of properties as function of pipeline depth.........................49
Table 3-4: Verification effort of "ALU Output" property for processors with asymmetric ALUs .............................................................52
Table 4-1: Microprocessor configuration..........................................................70
Table 4-2: Benchmarks..................................................................................70
Table 5-1: Latches necessary for the Success Monitor Switch............................104
Table 5-2: Target System Parameters..............................................................114
List of Figures

Figure 2-1: Comparison in coverage between design testing and formal verification ..... 16
Figure 2-2: Non-probabilistic versus probabilistic model checking ................................. 21
Figure 2-3: Comparison in coverage among simulation, exhaustive model checking, and bounded model checking ................................................................. 24
Figure 2-4: Translating signals between abstraction and implementation models through refinement maps ................................................................. 27
Figure 3-1: The abstract processor (ISA) as specification of correctness for the implementation processor ........................................................................................................... 39
Figure 3-2: The auxiliary state maintained for the abstract processor ................................. 42
Figure 3-3: The fragmentation due to having different L1 and L2 organizations .............. 47
Figure 3-4: Simple example of organizing the L2 cache to reduce the verification effort. 48
Figure 4-1: Traditional and proposed workflow for development of a new DPM scheme. ................................................................................................................................. 60
Figure 4-2: Example of trade-off analysis between verification-effort and performance. 61
Figure 4-3: DPM scheme with global controller ................................................................. 65
Figure 4-4: DPM scheme power utilization in time ............................................................. 66
Figure 4-5: Possible assignments of cores per controllers .................................................. 67
Figure 4-6: DPM code snippet in PRISM ................................................................. 72
Figure 4-7: The probabilistic model checking process .......................................................... 73
Figure 4-8: Impact of Number of voltage levels (VL) ......................................................... 77
Figure 4-9: Impact of number of frequency levels per voltage level (FL) ........................ 79
Figure 4-10: Impact of homogeneous policy ........................................................................ 81
Figure 5-1: Example of a power gateable circuit using a header transistor .................... 90
Figure 5-2: The break-even point.................................................................91
Figure 5-3: The possible consequences of deciding to power gate a functional unit .......92
Figure 5-4: Best-case potential to power gate for the SPECint (left) and SPECfp (right) benchmarks.........................................................................................................................94
Figure 5-5: Example benchmarks wasting energy with the IdleCount scheme............97
Figure 5-6: Efficiency counters........................................................................102
Figure 5-7: The token bag mechanism............................................................105
Figure 5-8: Interface between the power gating scheme and the rest of the system. The shaded block illustrates our proposed mechanisms: the Success Monitor Switch and the Token Counting Guard.................................109
Figure 5-9: Experimental framework ...........................................................113
Figure 5-10: Comparative energy savings for the benchmarks wasting energy under the IdleCount scheme. .................................................................................................119
Figure 5-11: Average energy savings (SPECint).............................................120
Figure 5-12: Average energy savings (SPECfp).............................................120
Figure 5-13: Worst-case Potential Performance Impact (SPECint). ......................122
Figure 5-14: Worst-case Potential Performance Impact (SPECfp)......................122
Acknowledgements

I would like to express my gratitude for the constant support that I received throughout my life in graduate school from my family, mentors, colleagues, and friends.

My fiancé, Costi, has shared with me these five years in graduate school and has helped me in innumerable ways. He gave me strength, always believed in me, listened to my ideas and brainstorming, and provided countless feedback and suggestions on writing and practice talks. My parents have provided me with a very supportive environment in which my curiosity and academic path were nourished and encouraged. My accomplishments are theirs as well.

I am grateful to my adviser, Dan Sorin for strengthening my motivation and for providing the intellectual guidance and mentorship that have been crucial in my development as a researcher. Dan has known how to cultivate my independence while at the same time giving me excellent supervision and advice.

I have been extremely fortunate to meet and be mentored for almost two years by Pradip Bose from IBM T.J. Watson Research Center. It was an absolute pleasure to work with Pradip during this time. He always found the right words to motivate and encourage me, and to challenge my intellectual curiosity with important and interesting research ideas. A large part of my thesis (presented in Chapter 4 “Verification-aware design for multi-core power management schemes” and Chapter 5 “Verification-aware design for unicore power management schemes”) was initiated during my two summer
internships at IBM T.J. Watson Research Center and was then continued at Duke University. At IBM I also had the great opportunity to work closely with and be mentored by Alper Buyuktosunoglu. Alper provided great mentorship, support, and advice, and I am grateful for learning a lot from him.

I am also grateful for receiving many insightful and helpful suggestions and advice from Alvin Lebeck throughout my graduate school. I am a better thinker and researcher because of Alvy.

My colleagues and friends have helped me in numerous occasions and in many ways. I am grateful to the Duke Computer Architecture group for their feedback on presentations and paper drafts as well as for the many interesting and challenging discussions during our reading group. My office mates and friends Fred Bower and Albert Meixner were a pleasure to be around, share tea with, and talk to, but also helped me directly with our computing infrastructure. I would also like to thank my colleagues and friends at Duke that have been a source of support and guidance for me – Shobana Ravi, Jaidev Patwardhan, Lavanya Ramakrishnan, Sukhendu Chakraborty, Laura Grit, and Aydan Yumerefendi.

Last but not least, I am very grateful to our graduate program coordinator Diane Riggs. Diane’s caring and supportive advice have made my transition to graduate school as smooth as possible. She always responded in the most helpful way to all my
questions, concerns, or needs throughout graduate school. My life in the department has been a lot easier because of Diane.
Introduction

Throughout the last decades the computing industry has fulfilled and even surpassed Moore’s law[62], which predicted a doubling of number of transistors fitting on a processor chip every two years. This increase in transistor density led to affordable computation power which permeated and brought tremendous benefits to almost all segments of our society, from the medical domain, to the car and airplane industries, financial markets, extreme weather prediction, etc. With changes in technology constraints the main processor design targets changed as well, from achieving the highest possible performance, to minimizing the die area, increasing reliability, and decreasing the overall power consumption. However, one constant requirement that remains throughout all technology generations is for computers to behave correctly, according to their specification and to produce the expected results.

At a high level, an incorrect behavior of a program can be due to either software or hardware causes. In this work we focus on the hardware causes of incorrect behavior which can vary from the presence of unmasked soft errors, to fabrication or wear-out hard errors affecting data or instructions, or to the exercising of a design bug. A significant amount of research has focused on evaluating the prevalence of soft [44, 51, 85] and hard errors [50] and on proposing techniques for preventing these errors from propagating into incorrect application behavior or state [12-14, 27, 67]. Ensuring design
correctness is mainly accomplished through the design verification step\(^1\). Not catching a soft or hard error at runtime is clearly undesirable as it may lead to an incorrect behavior of a specific processing machine. However, not catching a design bug in the design verification step can have catastrophic consequences because unlike soft errors, hard errors or fabrication bugs, design bugs affect all produced units. Documented correctness design bugs have led to processor recalls, as in the case of Pentium’s notorious floating-point division bug [9], with serious financial consequences for producing companies.

The same technology trends that sustain the unprecedented growth of the computing industry make processors design verification both more *critical* and more *difficult* to achieve. The steady decrease in transistor feature size has enabled embedded processor to become part of application domains like the airplane industry, the car industry, or the medical field, for which safety is truly critical. The presence of design bugs in processors used in these areas represents a safety hazard on a large scale. Increased difficulty of processor design verification is also indirectly caused by the availability of smaller transistors, because more complex processor designs can fit in the same area, resulting in additional state and control logic that need to be verified for correctness. An accelerated design cycle is an orthogonal factor that also contributes to

\(^1\) Another possible approach is to patch design bugs in the field [81, 82]. The advantage is that design bugs can be corrected in this manner in shipped products; the drawbacks are the requirements that the design bug must be identified a priori, detectable at runtime, and can cause a significant performance hit.
making the design verification more difficult through the reduction in the time available for the verification process.

The verification effort required for reaching desired verification targets is not a constant across designs but varies with the particular design’s characteristics. How easily verifiable a new processor design is can have a big impact on the value of that new design for the producing company. If the design’s verification difficulty is low, the verification can be accomplished in time for the new processor to be available during its scheduled market window and without consuming a tremendous amount of resources. In contrast, a processor design that is very difficult to verify can require a larger verification team than anticipated, can lead to missing the planned market window due to delays in reaching verification targets, and can even result in shipping a processor design with many design bugs if insufficient time was spent in verification.

Because of both its importance and difficulty, the design verification step currently consumes the majority of resources (60%-70%) [11, 37] allocated to the creation of a new processor. Furthermore, despite high resources allocated to it, the design verification problem is not solved, as suggested by dozens of design bugs present in the errata of recent processors shipped by major manufacturers [28]. Due to its high impact on the overall success of a new processor, design verification effort has become one of the critical resources in the design cycle. Since it is expected that Moore’s Law will continue to be fulfilled by the computing industry for at least a decade [32] it is
unlikely that the design verification problem will simplify by itself due to a decrease in processor state. The alternative for mitigating the design verification difficulty is to address it explicitly by considering verifiability a metric in its own right, to be targeted explicitly in the design of a new processor. It is this latter direction that we advocate throughout this thesis.

As is the case for other metrics of interest for architects, the decisions that have the largest impact on how verifiable a design is are the ones taken early in the design cycle. However, the current approach is that during the early stages of design architects mainly focus on making processors fast, power-efficient, and reliable, with limited explicit consideration of the impact these decisions might have on verifiability. Design verification is usually accomplished by a separate team, late in the design cycle. A scenario is thus possible where some decisions made early in the design cycle with the purpose of achieving a slight increase in performance end up complicating the verification to a large extent. This effect will be detected by the verification team only in the late cycle of design, when major changes to the design are very difficult to introduce.

Considering the critical impact that the verification step has on the design’s overall success we propose to consider the design’s verifiability from its initial conceptual stages and not only towards the end of the design cycle, when major changes are difficult to introduce. Consequently, in this thesis we advocate for including verifiability as a first class design constraint to be considered in the first stages of the
design cycle together with other established metrics like performance, power, and reliability. The intended effect of this verification-aware design is for designs to become more verifiable, such that the associated resources consumed by the verification process are decreased.

A similar transformation took place for power usage almost a decade ago [15-17]. The increasing number of transistors per core led to escalating values for total power consumption with all the associated negative consequences. However, once power became one of the critical design constraints (as it continues to be) it was no longer sufficient to consider it only late in the design, but power had to be explicitly included in the metrics evaluated early in the design cycle together with area and performance. A new series of designs were proposed that specifically addressed power constraints trading-off performance or area for the benefit of power [22, 23, 69].

There are multiple aspects to be considered for designing verification-aware processors. A verification-aware perspective can be applied to very different parts of the design such as the processor core, the memory hierarchy organization, the power management techniques, etc. What constitutes correctness across these areas may differ, which determines what kind of verification approach to use for a particular problem. A variety of verification techniques can be used for ensuring design correctness including testing, formal verification, as well as combinations of the two. The approaches differ in the type and amount of resources they require. Consequently, the metrics that can best
capture verification effort differ depending on the exact approach pursued for verification.

Clearly, the design space of modern processors is enormous and we don’t intend to evaluate the implications of verification on all aspects of processor design in this thesis. We instead select a representative subset of processor design applications and illustrate on this subset how the desirability of various design features changes when we promote verifiability to the status of a first class design constraint. We begin in the context of single core processors by investigating design choices for pipeline organization, operand bypass network, ALUs, memory hierarchy organization, and dynamic power management through power-gating. Multi-core processors are becoming increasingly prevalent and one of the most important design constraints in this design space is power management. This motivates us to evaluate verification-aware designs for dynamic power management techniques that target power capping with the use of dynamic voltage and frequency scaling. These different contexts allow us to gain insight into the implications of verification-aware designs across various domains, which have different correctness requirements best analyzed with different verification tools.

The rest of this chapter starts by presenting research that is related, at a high level to our proposal of verification-aware processor design. We then detail our thesis’ statement and primary contributions and provide an outline.
1.1 Research related to verification-aware design

There has been research that proposed changes in various design components in order to decrease the overall design effort or more specifically the verification effort. Other work has looked at the steps in the traditional verification process and researched ways to improve them individually. Throughout our thesis we also discuss other research related to specific design components we analyze.

1.1.1 Design for verifiability

Milne [60] first proposes the concept of “design for verifiability” as a means to improve the tractability of hardware verification. He advocates for having structural design rules in place to constrain the design’s behavior such that the difficulty of the design’s verification through a mathematical proof is decreased. The level of analysis in Milne’s paper is that of logic gates. Although in our work we make a similar argument of investigating changes in the design for the benefit of verifiability, our analysis is conducted at a different level of abstraction and scope, namely that of the microarchitectural design of a processor. The concrete proposal described in Milne’s paper is that of inserting latches in a design for decreasing the total possible states of the system compared to an asynchronous design. The design space we investigate - in terms of verifiability of various design options, trade-offs between verifiability and power and performance metrics - is thus necessarily different in our work due to the different level of abstraction and scope.
Curzon and Leslie [12] re-designed a switching fabric to be more easily verified with a theorem prover. After encountering difficulties in verifying an initial version of the switch through theorem proving, the person performing the formal verification proposed several design changes (increasing synchronization of data arrival events, restricting how certain bits are used, etc.) which were checked for estimated performance impact and approved by the original designer. Our work differs in several respects. First, we focus on processor designs and evaluate design alternatives and impact on verifiability in this specific context. Second, we construct a more precise quantitative metric to measure a design’s verifiability and evaluate trade-offs between verifiability, power, and performance goals. To quantify an increase in verifiability Curzon and Leslie mention the specific theorem proving targets which can be eliminated by a design change. Also, the formal verification techniques we use involve a larger degree of automation (largely model checkers and probabilistic model checkers) compared to the theorem proving used by Curzon and Leslie.

Another instance of redesign for verification is the work of Abdel-Hamid et al. [2] on the specification and verification of IEEE-754 table-driven exponential function. The authors observed that the initial design was flat which made formal verification through theorem proving intractable. The proposed redesign for verification structured the design into modules which the authors targeted to be self-contained for verification purposes and with reduced communicating signals between them. In our work we do
not focus on the verification of computation parts of the processor but on control flow and data flow. We also differ in the formal methods we use for verification (model checking versus theorem proving) and in our more precise quantification of a design’s verifiability.

BlueSpec [13] is a high-level, object-oriented hardware description language that can be compiled to both RTL and a term rewriting system for automated verification. BlueSpec eliminates bugs due to translating an RTL design into the language used for verification, and it is orthogonal and complementary to our work.

1.1.2 Complexity-effective design

Another research direction related, at a high level, to our proposal is complexity-effective design. There has been a significant interest in complexity-effective designs, including a workshop dedicated to this issue (Workshop on Complexity Effective Design). There are several definitions of complexity, most of which do not correspond directly to verification effort. Bazeghi et al. [5] developed the “μcomplexity” methodology for measuring and estimating processor design effort (without differentiating verification effort). The authors use real design effort data (in person-months) of modules from three processors to construct non-linear models for predicting design effort. In the attempt to find an accurate quantification of design effort, the authors find that the combination of number of HDL code lines and the sum of fan-ins of the logic cones in the design lead to a good metric, while the power, area, frequency,
number of flip-flops represent poor metrics. Other research, most notably that of Palacharla and Smith [68], has provided guidelines for designing scalable superscalar cores, and they dubbed this work complexity-effective design. Complexity, in the author’s terminology, is measured as the delay of the critical path through a piece of logic. However, what makes a structure “complex” in this interpretation can be quite different from what makes a structure difficult to verify. For example, the authors consider a large, associative structure to be much more “complex” than a small, direct-mapped structure. However, the effort required to verify a structure depends significantly on the symmetry of its entries. If a structure is symmetric, then increasing its size and associativity do not increase the verification effort (as we discuss in Chapter 3).

1.1.3 Improving individual steps in the verification process

Valeria Bertacco’s research emphasizes the difficulty of the verification process and the high motivation for achieving designs that function correctly under all possible execution scenarios. She proposes and pursues three research directions that would contribute to increasing processor’s correctness. In the first research direction she argues for the value of hybrid simulation-formal verification solutions [83] whose state exploration is guided by designer insight [76]. A second research approach targets to quantify the extent of the design space which was validated through simulation [82]. The third research direction builds on the second and takes advantage of the insight of
detecting at runtime which processor states have been verified. Semantic guardians are proposed that switch execution to a safe but less performant mode of operation whenever the processor reaches an unverified state [81]. All these research directions focus on improving the verification process without targeting to change the design such that it becomes easier to verify as we propose in this thesis.

1.2 Thesis Statement and Contributions

The goal of this thesis is to establish the validity of two related hypotheses: 1) making design choices with verifiability as a first class design constraint does decrease the verification effort necessary for establishing the correctness of the considered design; and 2) making explicit tradeoffs between verifiability, performance, and power enables designers to identify better design points for given verification, performance, and power goals.

In this thesis we make five main contributions:

1. We propose verifiability as a first class constraint to be considered together with performance and power in the early stages of processor design.

2. We evaluate the impact on verifiability of various choices for design parameters. In the context of single core processor design we analyze the organization of caches and translation lookaside buffers (TLBs), the depth of core’s pipeline, the presence of an operand bypass network, the type of ALUs used, and dynamic power management (DPM) schemes for energy reduction
through speculative power gating. In the context of multi-core processors, the design options we investigate are DPM schemes for power capping using dynamic voltage and frequency scaling (DVFS) techniques.

3. We analyze trade-offs between a design’s verifiability, performance, and power. More precisely, we evaluate DPM schemes through speculative power gating (for single core processors) and DPM schemes for power capping using DVFS (for multi-core processors).

4. We re-design several components for verifiability such as DPM schemes through speculative power gating (for single core processors) and DPM schemes for power capping using DVFS (for multi-core processors).

5. We propose design guidelines for increasing verifiability. In the context of single core processor design we discuss guidelines for the organization of caches and translation lookaside buffers (TLBs), the depth of core’s pipeline, the presence of an operand bypass network, the type of ALUs used, and dynamic power management (DPM) schemes for energy reduction through speculative power gating. In the context of multi-core processors our guidelines refer to DPM schemes for power capping using dynamic voltage and frequency scaling (DVFS) techniques.
1.3 Thesis Outline

This thesis continues with Chapter 2, which provides some background in verification through design testing and formal verification methods. This chapter describes verification methods and tools already available through advances in formal verification, a useful background for our research but not a contribution in itself. The following three chapters represent the core contribution of this thesis. They analyze verification-aware designs across different contexts: a processor’s core (Chapter 3), the dynamic power management schemes for multi-core systems (Chapter 4), and dynamic power management schemes for unicore systems (Chapter 5). Chapter 6 summarizes and concludes our thesis.
2. Background in Design Verification

The material presented in this chapter provides some background on the activities and methods in use during the design verification process. As design verification is a research field in itself, the information presented here is necessarily non-exhaustive and describes in short only the topics relevant for this thesis. Most of this chapter does not represent an original contribution. However, its content is very relevant in understanding the contributions of our thesis, which are described in detail in Chapters 3, 4, and 5 and summarized in Chapter 6. Section 2.6 is particularly important in this respect as it describes our proposed metrics for capturing verification effort. The quantitative analyses we present throughout this thesis, whether they compare multiple design from a verification perspective, or evaluate trade-offs between verification effort, power or performance, rely on the verification effort metrics presented in Section 2.6.

2.1 Design verification definition and approaches

Many activities are considered part of the verification effort of a processor design including design testing and formal verification. Design testing is the process of running a simulator of the processor, with either random or directed inputs for as many cycles as possible with the purpose of uncovering design bugs. The main disadvantage of design testing is its incompleteness, due to the impracticality of exhaustively testing a system
with as many possible states as a current microprocessor [6]. The advantage of testing the design through simulation is that the method is applicable to very large systems.

Formal verification is the act of proving or disproving that all possible behaviors of the system of interest comply with a formal specification of correctness. In contrast to design testing, formal verification has the advantage of being a complete verification solution, in that it examines, for ensuring correctness, all reachable states of a system. As we discuss below, the main drawback of formal verification is that it requires a large - sometimes prohibitive - amount of resources, either in manual effort or in memory and runtime.

Figure 2-1 shows a conceptual representation of the difference in coverage between design testing and formal verification. The red exterior circles in the figure depict the border on the reachable states of the system and every point within the circle represents such a reachable state. Both design testing and formal verification start from the initial state of the system and check that reachable states of the system are correct. However, design testing checks only the states of the system encountered in the simulated paths, while formal verification exhaustively checks every possible state of the system.
As depicted in Figure 2-1, formal verification has the clear advantage over design simulation that it exhaustively checks that every possible system behavior conforms to the specification of correctness. However, many systems of interest, including processor designs, are too complex to be fully verifiable through formal verification alone. In many instances formal verification can check only much simplified versions or subcomponents of the original designs. The formal verification step of the abstracted design is often followed by testing through simulation of a more detailed version of the design that is closer to the real system, but is also beyond the capabilities of formal verification tools. We believe that the current state of design verification that combines both formal methods with simulation will continue. Because of this state, throughout this thesis we consider the reduction in effort for both formal verification and simulation as important targets for a verification-aware design.

There are two main approaches for performing formal verification: model checking and theorem proving. In the following sub-sections we describe them both in more detail.
2.2 Formal verification through model checking

Model checking verifies that a model of the system complies with a desired correctness property by exhaustively enumerating all reachable states of the system and traversing the paths of execution through these states. There are two main characteristics which make the use of model checking for formal verification appealing. First, the process is fully automatic and because it does not require advanced mathematical knowledge from the user in proof building, it does not imply a steep learning curve. The second attractive characteristic of model checking is that in case the model violates the property under consideration the model checker will document the incorrect behavior by generating a counterexample starting in the initial state and ending in the violating state. The counterexample can then be used to debug the model. The exhaustive traversal of a model’s reachable states points toward the major drawback of this verification method. The size of the reachable state space of a system often increases exponentially as a function of size of the input set, which is also known as the state explosion problem. The result is that the resources required for model checking (in both memory and computation time) increase rapidly with the size of the model being formally verified, making the process computationally intractable even for models of modest size.

In order to obtain a proof of correctness for a system’s design through model checking three main steps need to be performed:
• Describe the model: The first step in model checking is to describe the system under verification in a finite state machine (FSM) description language. At this step, part of the system that is irrelevant to ascertaining correctness of the system can be abstracted away. The goal is to simplify the model such that it would be easier to model check.

• Specify correctness: A system is model checked with respect to a set of correctness properties. An important part of the model checking process is ensuring that the set of correctness properties that is being verified captures the entire set of behaviors as desired by the designer. If all correctness properties in the specification set are obeyed then the system is considered to be correct. If a desired property is missing from the specification of correctness, model checking will obviously not verify that the property holds and the verification process might produce a false negative. In case a correctness property fails, a counter example is generated.

• Perform the verification: As mentioned, one main appeal of formal verification through model checking is the automaticity of the process. Based on the FSM description, the model checking tool automatically explores all possible behaviors of the system and checks whether they obey the specification for correctness. However, the verification step does include some manual effort, although in a different way and to a smaller degree compared to theorem proving. If any of the correctness properties is not obeyed, the model checker will generate a counterexample starting
in the initial state and ending in the offending state. The counterexample needs to be investigated manually to reveal in which way the system behaved incorrectly. If the counterexample truly identifies an incorrect behavior pattern in the real design, the system and model need to be corrected. It is also possible for the counterexample to represent a false positive. This can occur when the error captured in model checking is in fact due to a poorly defined specification of correctness, or to abstracting away a relevant part of the real system behavior in the modeled FSM description. In such a case, the specification of correctness or the abstract model of the system need to be modified such that they do capture the correctness constraints and behavior of interest in the real system.

2.2.1 Non-probabilistic versus probabilistic model checking

The traditional and most widely used type of model checking attempts to identify whether an incorrect state of the system can ever be reached. The FSM description models all possible transitions of the system, without reconsidering how likely they are to happen. This type of model checking answers questions of the type “Can this incorrect behavior ever happen in the system?”. The Cadence SMV [42] tool that we use in the analysis of verification-aware designs at the level of the microprocessor core (Chapter 3) is an example of such non-probabilistic model checking.

However, for a certain category of systems, defining correctness in such absolute terms is too restrictive. Consider for example a power management scheme whose goal
is to cap the total power consumption below a set budget. Requiring that the power budget is never exceeded might be too restrictive, leading to pessimistic solutions that bring a high performance costs. It might be acceptable, from the designer’s perspective for the power budget to be exceeded, if a guarantee can be provided that the undesired behavior happens infrequently. Consequently, the correctness property to be verified for this system might be that, with a high probability, the total power consumption is maintained below the desired budget.

Probabilistic model checking permits the formal verification of such probabilistic properties. The difference from traditional model checking is that the system is described as a probabilistic FSM by specifying the probability associated with each transition of the system (the probabilities of all transitions from a certain state should sum to 1). Figure 2-2 illustrates this difference between a non-probabilistic FSM (left) and a probabilistic FSM (right). Correctness properties identify which states are illegal and should not be reached (the states represented as red circles).

Furthermore, in probabilistic model checking rewards (or tokens) can be associated with certain states (represented in the figure as small dots below the states). The rewards can be used, for example, to keep track of the power consumption or performance of the system when it reaches the specified state. Based on the defined transition probabilities, the model checker can estimate the cumulative rewards of the
system as it proceeds. The expected performance or power consumption of the system can thus be estimated in this way.

![Figure 2-2: Non-probabilistic versus probabilistic model checking](image)

### 2.2.2 Explicit state versus symbolic model checking

As mentioned, the main drawback of model checking is the prohibitive amount of resources required for the exhaustive traversal of the reachable state space. The first generation of model checkers, called explicit state model checkers required the allocation of memory linear in the global state graph of the system [26]. Since the size of the state graph of the system can be an exponential function of the state bits, only modest size systems could be formally verified through explicit state model checkers.

A breakthrough in formal verification through model checking took place when McMillan proposed *symbolic model checking* as an alternative to explicit state model checking [20, 56]. Symbolic model checking uses Ordered Binary Decision Diagrams (OBDD) [18] for representing the reachable state space of a finite state machine. The term *symbolic* refers to the fact that a single node allocated in memory could represent an entire set of reachable states of the system. For some systems, the OBDD representation
of the state graph is a lot more compact than the representation used in explicit state model checking. When the state space exhibits a certain regularity that the OBDD representation could take advantage of, it becomes possible to verify systems with an astronomic number of reachable states such as $10^{20}$ [20]. Later advances in model checking algorithms targeting representation through OBDDs enabled verification of even $10^{120}$ reachable states [19].

One caveat is that this reduction in number of nodes required to represent the state graph enabled by the OBDD representation is not guaranteed. It is possible that the number of symbolic nodes used in the OBDD representation is the same as the number of nodes required by explicit state model checkers, exponential in state bits. Furthermore, the number of nodes required by the OBDD representation depends not only on characteristics of the particular system modeled, but also on the order of binary state variables used in the OBDD algorithm. It is possible that, for the same system, a certain order of the state variables leads to the allocation of a number of symbolic nodes linear in the state variables, while a different order requires an exponential number of nodes. Heuristic algorithms have been developed for finding a good order of binary variables to be used by the OBDD representation; however, finding such an order is not guaranteed.

The memory requirements of explicit state model checkers increase as a linear function of the size of system’s state space, while the memory resources of symbolic
model checking are a function of number of nodes allocated in the OBDD representation (which can vary from linear to exponential in number of state bits). The Cadence SMV tool we use in analyzing verifiability of a processor core (Chapter 3) is an example of such a symbolic model checker that represents the reachable state space through OBDDs.

### 2.2.3 Complete versus bounded model checking

Regular model checking exhaustively traverses the entire reachable state space of the design and thus offers a proof that the system behaves according to the specification of correctness. For very complex systems, it is possible that performing complete model checking is intractable due to the amount of resources required. One option in that case is to perform bounded model checking [8], which unrolls the behavior of the FSM for $k$ number of steps from a specified state. Bounded model checking verifies whether a violation of any of the correctness properties can occur within $k$ steps. If such a violation is detected, a bug has been found and a counterexample is generated to document it. However, if no such violation can be found, no conclusion can be drawn on the correctness of the system. Bounded model checking is thus most useful for debugging purposes, when the system is guided through simulation towards a certain execution point, around which model checking is performed exhaustively for a limited radius. Figure 2-3 compares simulation with formal verification and bounded model checking in terms of coverage.
A similar unrolling of the system for a limited number of steps can be used in probabilistic model checking in the calculation of rewards. The PRISM probabilistic model checker [16, 15] that we use in our analysis of multi-core power management designs (Chapter 4) employs such a strategy for calculating the values of its reward structures. In our example the rewards are used for keeping track of power consumption and performance when the system functions under various power management schemes.

Figure 2-3: Comparison in coverage among simulation, exhaustive model checking, and bounded model checking

2.3 Formal verification through theorem proving

At a high level, theorem proving ascertains that a statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). Theorem proving can be performed manually, in a manner similar to general proofs commonly encountered in the domain of mathematics. An alternative is the field of automated theorem proving where the proof demonstrating that the conjecture can be logically deduced from axioms and hypotheses is accomplished by a software program interacting with the user [45, 66]. The biggest appeal of theorem proving is that it can,
and has been applied for formally verifying large and complex systems. For example, in the area of computer architecture, theorem proving has been used for the verification of pipelined processor designs with out-of-order execution, a reorder buffer, a store buffer, branch prediction, speculative execution and exceptions [34, 35]. The main aspect that discourages the use of theorem proving on a large scale for processor verification is the intense and highly specialized manual effort required for accomplishing the proofs. Also, in case the design changes, the extent and manner in which the proof needs to be modified are unclear. Furthermore, for an incomplete proof, little help is provided by the theorem proving tools as to what next step would bring the proof closer to the solution (in contrast to the counterexamples derived in model checking). These drawbacks of pure theorem proving techniques have prompted researchers to attempt to use theorem proving in combination with more automatic techniques like model checking. The main idea was to restrict the use of theorem proving to the minimum necessary for bringing the verification problem within the scope manageable through automated techniques. We describe several such techniques in the next section.

2.4 Combining model checking and theorem proving

In addition to pure model checking and pure theorem proving several tools achieve formal verification by targeting a synergy between the two approaches that would exploit the advantages of both worlds [7, 55, 57]. The use of theorem proving and model checking in combination enables a divide and conquer approach for managing
the complexity of the formal verification task. More precisely, theorem proving is used to decompose a larger problem into multiple properties whose aggregate is equivalent to the original verification problem [57]. Each of these properties is then discharged to a model checker for verification. For this approach to be successful, it is necessary for the decomposed properties to fit into the resources available to the model checker. The theorem proving step can be performed through either manual or automatic theorem proving. The Cadence SMV tool we use is an example of such a tool that combines model checking with automatic theorem proving. In the subsections below we discuss the main techniques available in Cadence SMV for decomposing the proof through theorem proving into smaller proof obligations manageable by a symbolic model checker [54].

2.4.1 Cone of influence reduction

For many correctness properties, whether the property is obeyed or not depends on only a subset of all state variables of the system. The cone of influence reduction takes advantage of this feature and performs a dependency analysis to determine all state variables that could potentially influence whether a correctness property is obeyed or not. The verification effort is reduced through this technique because resources are only allocated for the state variables that actually matter for deciding on the correctness of a property.
2.4.2 Refinement maps

For some systems the verification problem can be formulated as ensuring that a more detailed implementation of a system has a behavior that is equivalent to a more abstract model of the system. In this case, the abstract model represents the specification of correctness (“gold model”) for the target system to be verified (the implementation model). A refinement map represents a translation of signals between the specification and the implementation levels.

Figure 2-4: Translating signals between abstraction and implementation models through refinement maps

As illustrated in Figure 2-4, refinement maps can be used to describe the interface of a module of the implementation processor as a function of signals from the abstract model. The advantage of this approach is that proof obligations of the implementation model can be verified by using the input (environment) signals from the abstract model instead of the implementation one. Figure 2-4 also illustrates that in general the abstract model is simpler and has a smaller state space than the implementation model (the states are represented through small circles in both models).
If the environment variables are derived from the abstract model, through refinement maps, instead of the implementation model, the verification should be simplified.

### 2.4.3 Symmetry reductions

One key to reducing verification effort is to exploit symmetry among states (i.e., uncover states that are equivalent from a verification perspective) [38, 39]. The state space explosion problem can indeed be mitigated in Cadence SMV, to some extent, by exploiting symmetry [57]. Symmetry can significantly reduce the effective number of states that the model checker must traverse, because only one state among a group of symmetric states needs to be evaluated. For example, in a multiprocessor with 8 processors, the state in which Processor 1 has block B with Exclusive coherence permissions and the other 7 have no access to block B is symmetric to the other 7 states in which a single processor has B Exclusively and the other processors have no access to B. Symmetry can be exploited in two basic forms:

- **Structural symmetry** occurs when multiple components are equivalent and can be permuted without changing functionality. For example, there can be structural symmetry across cores in a multicore processor or across ALUs within a core. There can be structural symmetry across the entries of a buffer. For a system with N structurally symmetric components, the number of effective states that must be traversed by the model checker can be reduced by a factor of N factorial, because all states resulting from permuting the N components are equivalent.
• **Data value symmetry** occurs when a variable’s exact value does not matter. That is, we can choose any value from its range to represent this variable. An example of data value symmetry is a word of memory traversing the memory hierarchy. Data value symmetry is powerful because it can take a $2^b$ state space for a B-bit datum and reduce it to a single value.

Cadence SMV has a specific data type called *scalarset* for differentiating the variables that are symmetric. The user needs to identify the variables whose symmetry can be exploited for decreasing the effort of the verification process and declare these variables as being of scalarset type. To ensure that the variable is indeed used in a symmetric way in the design, several limitations are set in place by the tool on the operations allowed on a variable of scalarset type. For example, no constant values can be assigned to a scalarset and the only operation permitted is comparison with another scalarset variable.

**2.4.4 Temporal case splitting analysis**

Another approach for mitigating the state explosion problem is the use of case splitting analysis. Assume that a variable $v$ which can have one of three values $val1$, $val2$, and $val3$ at any point is involved in the cone of influence of a property $p$. In Cadence SMV is possible to decompose the verification of property $p$ into three sub-obligations each corresponding to one of the possible values of variable $v$. Since $v$ must have a value from its domain in every state of the system, the decomposition is complete. Temporal case splitting analysis can be applied for variables that are symmetric and thus are
modeled as a scalarset type. In this instance, major verification benefits can be achieved because it is possible that we only need to check one of the possible cases, the rest being reducible to it.

2.4.5 Data type reductions

Because model checking exhaustively traverses the reachable state space of a design, it can only be applied to finite state systems, or in other words designs that have all parameters of fixed values. However, in many cases it is desirable to verify a design where the values of some parameters are left unspecified. In the context of processor design for example it is useful to consider a cache or memory of unspecified size. The advantage of verifying such a parameterized design is that the same “verification” remains valid regardless of the precise values of the parameters in actual implementations.

The theorem proving layer present in Cadence SMV allows for such parameterized designs of infinite state spaces to be verified through a data type reduction method. The main idea is to rely on the theorem proving layer to correctly transform an infinite state system into a finite state one that is equivalent for verification purposes. This method works in combination with the symmetry reduction described previously. A scalarset variable can be declared as having an *unspecified* domain. When represented in the model checker layer, this variable will be interpreted as having a subset of actual values (for example 0, 1, and 2) and another symbolic value which is
NaN (not a number) and comprises the rest of the values. Having this data type reduction technique available brings clear benefits. However, the drawback is that the method relies on user guidance to specify how many real values should be considered when working with an undefined scalar set variable. When multiple variables of this type appear in the cone of a property the guidance required from the user can be significant.

### 2.4.6 Uninterpreted functions

Model checking is not particularly suited for verifying correctness of data processing units like the floating point unit for example. The reason is that it is often the case that verifying these units requires explicitly specifying the data bits and the precise operations that take effect on them. This is likely to degenerate into a state explosion problem. However, many systems of interest that are heavily control oriented, and thus suitable for model checking do have components that perform some specific operation on data. It would be desirable to still be able to model and verify the correctness of these systems but abstract away the exact operation taking place on the data. Cadence SMV uses the construct of *uninterpreted functions* for achieving that. An uninterpreted function is a simple mapping between a set of input variables to an output value and is represented in Cadence SMV through a single or multi dimensional array that maintains its values from step to step. All that can be said about an uninterpreted function is that if provided with the same input values it will return the same output value. For example,
if an uninterpreted function is used for modeling an ALU unit, the uninterpreted function makes it possible to check that the value stored in a register after an ALU operation was not corrupted by comparing it with the value of the uninterpreted function.

### 2.5 Objectives of using formal verification

We identify two different benefits of employing formal verification early in the design cycle. The first and most direct benefit is to obtain a formal proof of system’s correctness. This approach can be pursued when a system model which captures the desired level of detail does not require a prohibitive amount of resources to be formally verified.

A second motivation for applying formal verification in the overall design cycle is for gaining insight into which of the design parameters scale well in terms of verification effort. This approach would be useful when a model of the system with the required level of detail cannot be successfully verified by formal verification due to excessive resource requirements. It is also possible that a different type of modeling from what is possible in formal verification tools is required for ascertaining correctness. For example, power management techniques require low level power models to ensure that the power budget of the system fits the envelope. Even when additional verification is necessary after the formal verification step, it is still valuable to have insight into
which design parameters are the ones that most contribute to making the system difficult to verify.

### 2.6 Metrics for capturing verification effort

Quantifying verification effort is not straightforward. We differentiate the metrics to be used depending on how correctness of the system is ensured and on the main goal of using formal verification.

If a formal proof of the system is obtainable through formal verification, the verification effort metrics should capture the effort required by the formal verification tool used. There are usually two components that go into the formal verification tool effort: manual effort and resource usage (memory consumption). The manual effort is mostly related to the theorem provers, while resource usage usually limits the model checkers.

As mentioned, one of the tools we use is Cadence SMV which contains both a model checking and a theorem proving layer. The manual effort is largely related to correctly guiding the theorem proving layer in decomposing the proof of correctness in sub-obligations which can then be discharged to the model checker. We found that manual effort is a function of both the number of properties that must be verified and the number of cases per property. The user is responsible for defining all of the properties and, for each property, identifying all of the possible cases that must be proved. In particular, we have found that case splitting is a difficult process. For
example, in our experience it was far more work to guide the theorem prover to verify a system with 2 properties each of which has 10 cases than to verify a system with 10 properties each of which has 2 cases. We use the number of properties and the number of cases per property as our metrics for theorem prover effort, even though they are imperfect.

For measuring resource consumption effort, we could consider the verification effort to be directly related to the number of state variables in the system. As the number of states increases, the memory consumption increases and eventually the model checker runs out of memory and hangs. This would be an appropriate metric for explicit state model checkers which assign memory for each reachable state. However, for symbolic model checkers, the memory usage is actually proportional to the number of OBDD nodes allocated. The number of OBDD nodes allocated depends in turn on the particular ordering of state variables used. It is possible that Design A has more state variables than Design B, but the model checker just happens to choose a more fortuitous ordering of state variables for an OBDD in the model of Design A and thus uses less memory. To capture this we use both number of state variables and number of OBDD nodes for measuring resource effort.

The second objective of using formal methods that we considered is to gain insight into which of the design parameters are contributing most to difficulty in verification. That insight could then be used in making verification-aware design
decisions that affect verification effort even for verification steps additional to formal methods (simulation). To capture verification effort in a manner that is relevant across multiple levels of verification we chose to use the *number of reachable states* as a verification effort metric. We believe the number of reachable states of the system accomplishes the goal of being closely related to verification effort both in formal verification and in simulation.

In the following sections that describe in more detail three design areas that we investigated for verification-aware design we mention the objective of performing formal verification and metrics used.
3. Verification-aware design for processor’s core

The overall goal of the research presented in this chapter is to analyze the impact of several processor design choices on formal verification effort. More specifically our objectives are to identify which of the design parameters analyzed make verification though formal methods difficult, and to propose a set of guidelines for decreasing the verification effort.

Using Cadence SMV, a composite formal verification tool that combines model checking and theorem proving, we explore several aspects of processor design, including caches, TLBs, pipeline depth, ALUs, and bypass logic. To start, we describe the processor design we modeled as well as the verification tool and methodology we used. We then present our results for several case studies investigating the impact of mentioned design choices on overall verification effort. We show that subtle differences in design decisions can lead to large differences in required verification effort. In conclusion we propose a set of design guidelines and an example of redesign for verification.

3.1 Formal verification methodology

As foreshadowed in the introduction, in order to understand the verification effort and the design characteristics that influence a design’s verifiability one needs to take into account the specification of correctness for that design, the goal of using formal
methods in the verification process, and the specific formal verification tool used. This section details these aspects in the context of processor core verification.

### 3.1.1 Objective of using formal verification

We consider that the main goal of using formal verification in the context of the processor design features analyzed in this chapter is to obtain a proof of system’s correctness, not to provide information for ulterior verification steps (e.g. through simulation). As such, the metrics we use for quantifying the verification effort capture resources required by the formal tool we are using (Cadence SMV). As mentioned in Section 2.6 these metrics are: state variables, OBDD nodes, case splits per property, and the total number of properties.

### 3.1.2 Formal verification tool

As mentioned in Chapter 2, the formal verification tool we used for our analysis of processor core design features is Cadence SMV, a formal tool which combines model checking with theorem proving. Our choice was motivated by the plethora of techniques available in Cadence SMV (described in more detail in Section 2.4) for mitigating the state explosion problem such as: cone of influence reduction, refinement maps, symmetry reduction (structural and data valued), temporal case splitting, data type reductions, and uninterpreted functions. We took advantage of all of these features when formally verifying our processor design model.
3.1.3 Specification of correctness

We ensure that the implementation processor functions correctly by proving that its behavior is equivalent to that of a very abstract and trivially correct version of a processor, which we call the abstract processor. This abstract processor constitutes the specification of correctness, or the gold standard for the verification of the more complex implementation processor (Section 3.1.4).

The abstract processor is a single cycle, in-order processor that has no caches, no TLB and performs no branch prediction. It can execute the same set of instructions as the implementation processor namely: an ALU-type instruction, Branch, Load, and Store. For the Load and Store memory operations the abstract processor first accesses the page table and then the main memory directly.

3.1.4 Processor model

The processor we are formally verifying for functional correctness has a 5-stage (Fetch, Decode, Execute, Memory, and Writeback stages), in-order, scalar pipeline. The ISA supported is formed of the following instructions: an ALU-type instruction, Branch, Load, and Store. The processor has branch prediction with a predict not-taken scheme. Its memory system has a physically addressed, single-level, write allocate cache and a TLB.

For the Load and Store memory operations the processor first accesses the TLB to calculate the physical address. On a TLB miss, the processor then accesses the page
table. With the resulting physical address, the last step is to attempt to access the cache or, in case of a cache miss, the main memory.

We call the target processor model that we are interested in formally verifying the *implementation processor*.

### 3.1.5 Verification process

Figure 3-1 illustrates the equivalence verification process. The dotted arrows in the figure link variables in the implementation processor with variables from the abstract processor. These represent refinement maps that were discussed in more detail in Section 2.4.1. For example, an arrow labeled PC means that the value of the program counter in the implementation processor should be the same as the value of the program counter in the abstract processor. The verification process translates into proving that the abstract and implementation processors behave in such a way that all refinement maps are obeyed.

![Figure 3-1: The abstract processor (ISA) as specification of correctness for the implementation processor](image)

39
Table 1 enumerates the properties to be verified. The properties’ names designate which variables must have the same values in the abstract and implementation processor. For example the refinement map Instruction ascertains that the abstract and implementation processor fetch the same instruction.

Table 3-1: Properties to verify.

<table>
<thead>
<tr>
<th>Fetched Instruction Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter, Next Program Counter, Instruction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decoded Instruction Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode, Source Registers, Destination Registers, ALU Operands, Branch Operands</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Access Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Address, Load Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Result Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU Output, Branch Target, Branch Condition</td>
</tr>
</tbody>
</table>

The abstract processor completely executes an instruction per cycle, while during the same cycle the implementation processor can move all instructions in the pipeline one stage further. When the implementation has to stall or recover from misspeculation, the abstract processor stalls as well; except in these ‘recovery’ cycles, the abstract processor is always synchronized with the implementation processor’s first pipeline stage.

As Figure 3-1 shows, the refinement maps link equivalent states in the implementation and abstract processors from all pipeline stages. Verifying correctness of the refinement maps which connect state in the abstract processor to state in the first
stage of the pipeline of the implementation processor is relatively straightforward. The formal tool compares the current state of the abstract processor to the current state of the implementation processor. This approach can be pursued because, as mentioned, the abstract processor’s state and the implementation processor’s first pipeline stage are synchronized.

To verify the correctness properties of stages deeper in the pipeline than the first stage, we need to ‘remember’ (maintain) the correct execution state of the abstract processor for the instruction currently in that pipeline stage. If this information is not maintained explicitly, the abstract processor progresses to the next instruction and the correct values against which the implementation state needs to be compared are lost. Cadence SMV has the capability to store state as auxiliary structures. The operations that can be performed on state declared as auxiliary structures are limited to value assignment and comparisons. We take advantage of this feature for storing the state resulting from the correct execution of an instruction by the abstract processor. Figure 3-2 illustrates this process. Another way to read the labels “Auxiliary State 1”, “Auxiliary State 2” from Figure 3-2 is: the state of the abstract processor one cycle ago, two cycles ago, and so on. This interpretation of the labels also provides some intuition on why verifying properties involving operand values for later stages in the pipeline involves more state. More auxiliary variables are included in the cone of influence of these properties.
Figure 3-2: The auxiliary state maintained for the abstract processor.

3.2 Case studies in processor core verification

This section presents the results of experiments where we change the processor design in four different ways and examine the implications for verification effort. One of the design characteristics that we thought promising to investigate from the perspective of verification effort was symmetry. If a system has symmetric entities, performing any permutation between these entities leads to no change in the semantics of the system. For such a system, the configurations obtained through permutation are equivalent and thus only one of them needs to be checked for correctness, which can potentially decrease the required verification effort. Based on the hypothesis that presence of symmetry can have a significant impact on verifiability effort, we investigated two design options where symmetry is affected: the first experiment inspects associativity and replacement policies for Caches and TLBs, and the second compares homogeneous and heterogeneous ALU units. We also analyzed the impact of the pipeline’s depth and pipeline’s bypassing network on verification effort.
3.2.1 Cache and TLB organization (associativity and replacement policy)

One of the basic design issues for a cache or TLB is choosing how set-associative it will be. The well-known engineering tradeoff is that increasing the associativity reduces the miss rate at the cost of being slower and more power-hungry. But how does the choice of associativity affect verifiability? We investigate the effect of associativity for caches and TLBs in this case study. Our hypothesis was that having a fully associative organization for the caches and TLBs should result in the highest level of symmetry that can be exploited in the process of formal verification.

Intuitively, a fully-associative structure is the most symmetric, because all of its entries are equivalent. This symmetry reveals itself when we describe an address in a fully-associative cache. It consists of a tag and a block offset, both of which can be represented by scalar sets (SMV’s symmetric type of state variable). Note that we cannot treat the address as a single scalar set, because sometimes we operate on only part of an address (e.g., when comparing tags). A k-way set-associative structure has less symmetry than a fully-associative structure, because each address must be treated as a concatenation of three scalar sets instead of two. There are now scalar sets to represent the tag, index, and offset. Thus we have more state variables (and a larger state space) that the model checker must consider every time that it encounters an address. A direct-mapped (1-way) structure has the same symmetry as a k-way structure, because it still requires three scalar sets to represent an address.
To quantify the verifiability impact of associativity, we used SMV to verify processors with caches and TLBs with various associativities and a random replacement policy. For clarity, we present data for only three scenarios: fully-associative cache and TLB, $k$-way set-associative cache and TLB, and direct-mapped cache and TLB. Note that the size of each structure does not impact the verification effort for any property, because the number of sets is represented as an unbounded scalar set.

In Table 3-2 we show the verification effort for all properties as a function of the cache associativity. The shaded entries denote the properties that are not affected by cache associativity. We observe three interesting phenomena, and we use bold text to highlight the results for three of the most affected properties in the table. First, we conclude that our hypothesis was correct—in general, a fully-associative structure requires less verification effort than a set-associative structure. This result may not be relevant to most caches, because we cannot implement large fully-associative caches, but it may be important in other points of the design that require smaller cache-like buffers. Second, we see that the results for $k$-way set-associative are almost always equal to the results for direct-mapped. Also, for two properties, the number of cases is actually slightly greater for the $k$-way structures than for direct-mapped and fully-associative. For these properties (Load Value and Physical Address), we had to consider an additional case depending on which way of the set-associative structure is used. Fourth, we observe that the number of OBDDs is generally related to the number of state variables,
but it is not a perfect correlation, for reasons discussed in Section 2.6. In fact, there are some properties which have zero state variables, but still have a non-zero number of OBDD nodes. These situations occur when verifying the property involves combinational variables but no state variables.

Table 3-2: Verification effort of properties as function of cache and TLB associativity.

<table>
<thead>
<tr>
<th>Property</th>
<th>State Variables</th>
<th>OBDD Nodes</th>
<th>Case splits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>full/</td>
<td>k-way/DM</td>
<td>full/DM</td>
</tr>
<tr>
<td></td>
<td>k-way</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>full</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>full/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>k-way/DM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>k-way/DM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DM/DM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DM/DM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DM/DM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                   | full/           |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | full/           |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |

|                   | full/           |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | full/DM         |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | k-way/DM        |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |
|                   | DM/DM           |            |             |             |         |         |       |

In a related set of experiments we evaluated the impact of cache and TLB replacement policies by comparing verification effort for random, Not Most Recently Used (nMRU), and Least Recently Used (LRU) policies. Our hypothesis was that the LRU policy would be hardest to verify due to restricting dependencies imposed between otherwise symmetric ways in the buffers. However, the results showed no significant
verifiability difference across the three policies, with random and nMRU involving basically the same verification effort and LRU only marginally more. To model the nMRU case, a variable holding the most recently used entry was added for each set, which translated into an additional number of state bits logarithmic in the number of ways (not a significant increase). For the LRU case, each way had a counter that tracked the time when a particular entry was last accessed. The reason why the additional counter did not considerably increase the verification effort is due to the cone of influence reduction technique available in SMV. Proving an individual property involves only a limited number of entries in the cache (not the extra state in the entire cache), which restrained the impact on the verification effort.

3.2.2 L1 and L2 Cache Organizations

If in the previous section our analysis focused on processors with a single level of cache, in this section we evaluate processors with two levels of caches. More precisely, we investigate how different L1 and L2 cache organizations can impact the verification effort. In general, the L2 cache is almost always significantly larger than the L1 caches, and it may have a different associativity or block size. These differences between the cache levels lead to differences in how addresses are interpreted and can thus decrease symmetry. In Figure 3-3 we illustrate how differences between the L1 and L2 organizations can force SMV to use five scalarsets to represent each address.
One easy way to improve verifiability is to maintain the same block size between cache levels, instead of having larger block sizes in the L2 than in the L1. That would reduce the number of scalar sets per address from five to four. The tradeoff is that we lose the ability to exploit more spatial locality in the L2.

![Diagram](image)

**Figure 3-3: The fragmentation due to having different L1 and L2 organizations**

A perhaps less obvious way to improve verifiability is to set the L2’s associativity to be K times more than the L1’s associativity, where K is the ratio of the size of the L2 to the L1. By organizing the L2 in this fashion, we can then interpret addresses the same way for the L1 and the L2. That is, the same bits are used for the tag, index, and offset, as shown in Figure 3-4. In the figure, the L1 and L2 have the same block size, so they have the same number of block offset bits. The L1 is 2-way associative and has 4 entries total, so it has 2 sets. The L2 is 8-way associative and has 16 entries total, so it also has 2 sets. We chose the L2 to be 4 times more associative than the L1 because it has 4 times as many entries, and thus they have the same number of index bits.

With this approach, we reduce the number of scalar sets per address to three. The tradeoff is that we have restricted the organization of the L2, and it may not be feasible to organize it this way. If the L1 is small and highly associative (e.g., 8KB, 4-way) and we
want an L2 that is much larger (e.g., 2MB), then the L2 would have to be prohibitively associative (1K-way!).

![Diagram of cache organization](image)

**Figure 3-4:** Simple example of organizing the L2 cache to reduce the verification effort.

### 3.2.3 Pipeline depth

In this line of experiments the leading hypothesis was that having more state in flight at the same time would be detrimental for formal verification effort. Although this expectation is an intuitive one, we wanted to quantify this effect, particularly considering that the state included in the cone of influence varies with each property and does not include the entire system.

To analyze this hypothesis we measured verification effort necessary for three different pipeline depths. The processor models compared implement a 4-stage (FD, X, M, W) pipeline, the classical 5-stage (F, D, X, M, W) pipeline as well as a 6-stage pipeline that has an extra stage for memory access. The memory system is fixed in all experiments; the cache and TLB are set-associative with random replacement.

In Table 3-3, we show the results of verifying all of the properties, and we use bold text to highlight a subset of the properties that reveal the most significant trends.
Table 3-3: Verifiability of properties as function of pipeline depth.

<table>
<thead>
<tr>
<th>Property</th>
<th>State variables</th>
<th>OBDD Nodes</th>
<th>Case splits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Depth 4 (D4)</td>
<td>D5</td>
<td>D6</td>
</tr>
<tr>
<td>PC</td>
<td>10</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Next PC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Instruction</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Opcode</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Source Registers</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Destination Regs</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>ALU Operands</td>
<td>22</td>
<td>29</td>
<td>37</td>
</tr>
<tr>
<td>Branch Operands</td>
<td>18</td>
<td>26</td>
<td>34</td>
</tr>
<tr>
<td>Load Value</td>
<td>30</td>
<td>41</td>
<td>52</td>
</tr>
<tr>
<td>Physical Address</td>
<td>12</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>ALU Output</td>
<td>9</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Branch Target</td>
<td>0</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Branch Condition</td>
<td>7</td>
<td>23</td>
<td>23</td>
</tr>
</tbody>
</table>

The shaded entry emphasizes the property for which its verification effort is almost unaffected by the depth of the pipeline. As can be observed from Table 3, the depth of the pipeline has a significant impact on the verification effort for most properties. Furthermore, there is variation in the effect an increase in pipeline depth has on the verification of different properties. This variation effect can be explained by considering the synchronization of the abstract and implementation processor described in Section 3.1.3. Adding pipeline stages in the beginning of the pipeline (as when going from the 4-stage FD, X, M, W to the 5-stage F, D, X, M, W) impacts more properties than adding a stage later in the pipeline (as when adding an extra memory stage for the 6-pipeline processor). This is an interesting result, since it points out that even for systems
having the same number of pipeline stages, the verification effort necessary for proving correctness can be different depending on functionality of the stages.

Another observation that can be made from Table 3-3 concerns the last column of the table. Varying the size of the pipeline leads to no change in the number of cases for each property. The intuition behind this is that additional stages do not change the path through the system that needs to be considered for each property.

Although the absolute number of state variables for each property is well within the model checker’s capability, this number would be far greater for a more complicated processor with more components and interactions between them. This simple implementation processor is likely a “best-case” scenario, in that the verifiability of a more realistic processor model would likely be even more dependent on pipeline depth. For a more complicated processor, we would expect to need to offload model checker effort to theorem prover effort, and we would have far more properties and cases. Our results support the hypothesis that having multiple cores with shorter pipelines (e.g., Sun’s Niagara [46] or Piranha [4]) is beneficial for verifiability, as opposed to super-pipelined processors (e.g., Pentium4 [10]).

3.2.4 ALU specialization

Symmetry in a design can exist at the level of modules when these modules are identical and interchangeable. We hypothesized that a core design that has heterogeneous, specialized ALU units would be more difficult to formally verify than a
design with homogeneous units. To analyze this hypothesis, we conducted an experiment that compares three different implementation processors. The first processor has only one type of ALU instruction and one type of ALU unit. The second processor has three different types of ALU instructions and one type of ALU unit. The last processor has three different ALU instruction types and three different ALU units, one for each type of instruction.

The results from this experiment showed that our hypothesis was not supported. Additional heterogeneous ALU units do not significantly increase the amount of state necessary for the verification of any single property. The explanation is in the way the theorem proving layer verifies correctness properties. Having more types of ALU instructions and ALU units means that the theorem proving layer will verify separate cases depending on the type of instruction encountered. The total amount of state per case will not change, there will only be more cases verified separately by the model checker layer. In other words it is possible for the theorem proving layer to decompose the complexity added through the presence of heterogeneous ALUs because not all ALUs are used at the same time in executing one instruction. By verifying separately that each type of ALU instruction is correct the verification effort is not composed.

The results are summarized in Table 3-4, which presents data for a single property checking that the result of the ALU unit is correct (the other properties have similar results). The small difference in State Variables between the first case and the
other two is due to an increase in instruction types which results in more bits being necessary to encode the opcode (3 instead of 2).

Table 3-4: Verification effort of "ALU Output" property for processors with asymmetric ALUs

<table>
<thead>
<tr>
<th>Processor type</th>
<th>State vars.</th>
<th>OBDD nodes</th>
<th>Case splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 type ALU instructions, 1 type ALU unit</td>
<td>28</td>
<td>10771</td>
<td>5</td>
</tr>
<tr>
<td>3 types ALU instructions, 1 type ALU unit</td>
<td>30</td>
<td>10462</td>
<td>5</td>
</tr>
<tr>
<td>3 types ALU instructions, 3 type ALU unit</td>
<td>30</td>
<td>10462</td>
<td>5</td>
</tr>
</tbody>
</table>

3.2.5 Operand Bypass Network

One other hypothesis that we investigated was whether the bypassing network that can be present in the design of a pipelined processor increases the verification effort. Architects have long considered bypassing to be a significant source of design complexity, and it would also appear to aggravate formal verification by adding to the number of paths that data could take through the pipeline. The bypassing network affects the dependence between variables in different stages. One possible consequence of this change in dependences is an increase in the cone of influence of properties and thus in the verification effort. To experimentally investigate this hypothesis, we compared two processor models. Both have a 5-stage pipeline with a k-way associative cache and a TLB with random replacement policies. Processor1 bypasses operand values between stages while Processor2 does not, stalling to solve dependencies of values in the pipeline (until these values become available through the register file or memory). The results, which initially surprised us, showed no significant difference in verification
effort for the two processors. At a high level, the explanation is that the information necessary for determining when to stall due to dependencies is the same as the information involved in bypassing of values. The properties have the same cone of influence in both cases, with only minor differences in combinatorial variables allocated.

### 3.3 Guidelines for verifiability

In light of the cases discussed above, we summarize two higher-level guidelines for decreasing the verification effort of processor models.

**Guideline 1.** Try to avoid performing operations on only a part of an otherwise symmetric field. Fragmentation decreases the symmetry we can exploit and translates into using more state variables to model entities of that type which in turns increases the overall verification effort.

**Guideline 2.** Try to limit the depth of pipelines. Having more pipeline stages translates into an increase in state that needs to be evaluated during model checking. This is particularly true for establishing correct functioning of later stages in the pipeline.

### 3.4 Related Work

In this section we discuss research that proposes modifications of the processor’s design for the benefit of verification.
3.4.1 Design guidelines for verifiability

Martin [52] investigates the verification complexity of cache coherence protocols. He argues for two rules that would benefit the formal verification of a multiprocessor: a) separating the enforcement of cache coherence from memory consistency and b) separating cache coherence protocols from properties of the interconnection network. The high level qualitative discussion argues that following these guidelines would make the design more modular and thus easier to verify. From this perspective, token coherence and directory coherence protocols should be easier to formally verify compared to snooping protocols. In the same realm of verifying cache coherence protocols, Marty et al. [53] argue that token coherence protocols are at an advantage, for verification purposes, over directory and snooping protocols because they separate correctness from performance. The quantitative comparison of coherence protocols verification effort was done by referring to the number of non-comment lines in the protocol model descriptions.

3.4.2 Dynamic verification

DIVA [3] proposed using a small, provably-correct checker core to dynamically verify a large, complicated superscalar core. Their observation is that it is impossible to statically verify the superscalar core, but they can achieve the same result by verifying its execution at runtime. DIVA can dynamically verify all portions of the superscalar processor whose functionality is replicated in the checker core. By definition, dynamic
verification cannot verify that a design is correct; rather, it verifies that a particular execution is correct. A dynamic verification approach like DIVA is fundamentally different from schemes that use homogeneous redundancy (e.g., redundant processors or threads) to tolerate faults, because identical components will share the same design bug and thus not be able to detect these bugs. Some later research developed dynamic verification mechanisms for a multiprocessor’s memory system, via runtime checking that it obeys coherence [24, 78] or memory consistency [59]. Meixner et al. [58] research dynamic error detection solutions specifically for simple cores. They demonstrate that correctness of execution of any von Neumann core can be inferred from correctness of control flow, dataflow, computation and memory access, and they propose dynamic checkers for each of these tasks. The above research differs from our thesis through its focus on dynamic verification solutions as opposed to static, design time verification.

3.5 Conclusion

In this chapter, we have argued that architects should explicitly consider verifiability when designing processors. The resources devoted to design verification are too expensive for us to continue to design processors without treating verifiability as a first-class design constraint, just like performance, power, temperature, reliability, etc. We have shown that subtle differences in a microarchitecture can lead to significant differences in verification effort. Looking into the future, we believe that the high costs of verification will motivate more radical redesigns of processors in order to ease the
verification process. Tiled architectures like TRIPS [21], Raw [84], or WaveScalar [79] offer the potential for reduced verification effort, because the verification of each tile is the same, although we still must verify that the tiles communicate and synchronize correctly. Even when designing a tiled architecture, though, we believe that architects must still be careful to avoid adding features that complicate verification despite only offering limited benefits in other metrics.
4. Verification-aware design for multi-core power management schemes

The prevalence of multicore architectures coupled with demands for low power systems motivate the development and evaluation of efficient power management solutions targeted specifically at multicores. Power is managed for several reasons, including to: improve power-efficiency, avoid power spikes, increase battery life, reduce the cost of providing power to the chip, and manage temperature. In this chapter, we investigate dynamic power management (DPM) schemes that can cap the peak power usage of a multicore. Providing a DPM scheme that caps the peak power can relax the power constraints placed on other system components or it can reduce overall costs by decreasing the system’s cooling costs or by matching system’s energy consumption with energy price throughout the day.

One important challenge for multicore DPM schemes is verifying that they are both safe (meet their power goals) and efficient (achieve as much performance as possible without exceeding power constraints). The verification difficulty varies among designs, depending, for example, on the particular power management mechanisms utilized and the algorithms used to adjust them. However, verification effort is often not considered in the early stages of DPM scheme design, leading to proposals that can be extremely difficult to verify.
To address this problem, we propose using formal verification (with probabilistic model checking) of a high-level, early-stage model of the DPM scheme. Using the model checker, we estimate the required verification effort, providing insight on how certain design parameters impact this effort. Furthermore, we supplement the verifiability results with high-level estimates of power consumption and performance, which allow us to perform a trade-off analysis between power, performance, and verification. We show that this trade-off analysis uncovers design points that are better than those that consider only power and performance.

Our main contributions are the following:

- We propose the use of verification effort as an additional metric to be considered, together with performance, in the early stages of DPM scheme design.
- We investigate and compare the scalability of effort necessary to verify different power management policies as a function of the available DPM mechanisms, such as the number of voltage and frequency levels and numbers of cores assigned to a controller.
- We evaluate the trade-offs between required verification effort, efficiency, and safety of particular DPM schemes.

### 4.1 Formal verification methodology

We begin this section with our objective in using formal verification, the motivation behind using probabilistic model checking to verify the analyzed DPM
schemes, and a brief overview on this method. Then we provide details on the particular methodology we use to conduct our experiments.

4.1.1 Objective of using formal verification

One critical aspect in the development of a new DPM scheme is its verification. As a concrete example of the importance of DPM verification, concerns over Intel’s Foxton DPM scheme [70] led to it being disabled in the first Montecito chips [31]¹.

The current industrial workflow in the development of a new DPM scheme is illustrated in the left side of Figure 4-1. At an early stage, the focus is restricted to maximizing the efficiency of the DPM scheme, with limited consideration of its verification. Later, the scheme is implemented in detailed, low-level simulators, and verification primarily checks whether the scheme achieves its efficiency goals. The clocks depicted in the Figure 4-1 symbolize design time; the larger the clock associated with a particular design step, the longer its required design time.

The problem with this current workflow is that it is prone to missing bugs. First, simulation is by definition incomplete as a verification solution, because only the states that are reached in a particular simulation path are ascertained to be bug-free. Second, if verification feasibility is not considered at design time, the reachable state space of the resulting DPM scheme can be enormous, which is problematic. Workflows often have goals for achieving minimum coverage (as illustrated in Figure 4-1), so having more

---

¹ Intel has not officially stated whether the concerns were over safety, efficiency, or functionality bugs.
states requires more simulation cycles and time (suggested through the clock symbol in Figure 4-1). If no coverage goal is specified, having more states increases the probability that undiscovered bugs remain in the design and decreases confidence in DPM correctness.

Figure 4-1: Traditional and proposed workflow for development of a new DPM scheme.

To address the above concerns, we propose the introduction of an additional, early step in the development of a new DPM scheme. We illustrate this added step in the right portion of Figure 4-1. This additional step creates, at an early design stage, a high-level model of the proposed power management policy which is then verified for efficiency and correctness using probabilistic model checking, an exhaustive formal verification method. By performing a high-level verification early in the development
process we have several advantages. First, we can identify problems when they are easier to solve. Second, a high-level model is also much easier to develop and modify than a detailed simulator, so we can quickly explore numerous designs. Third, we can perform an early trade-off analysis between verification-effort and other metrics of interest to DPM design (such as efficiency and safety) and guide the design towards verification-aware solutions. An illustration of this trade-off evaluation process is shown in Figure 4-2, where we analyze which value of a DPM parameter should be used in the design. While a larger value of the parameter does lead to a slight increase in performance, the benefit might not be worth the large additional verification cost.

![Diagram](image)

**Figure 4-2: Example of trade-off analysis between verification-effort and performance.**

The benefit obtained from designing a system that is easier to verify is illustrated in Figure 4-1 through a decrease in the size of the clocks associated with design iterations due to verification aspects. Specifically, for an easier to verify design, design bugs would take less time to find and also desired coverage goals would be reached sooner.
For an accurate evaluation of the role formal methods can play in the verification of DPM schemes, we want to note that DPM schemes can be verified in a model checker only at an abstracted and high level. The analytic power models and performance models that can be used - and not trigger state explosion problems in model checking - are informative but also necessarily simple. Thus, model checking does not eliminate the need to later simulate a detailed implementation of the DPM scheme (as shown in Figure 4-1). However, using model checking of high level DPM schemes can help catch design bugs early, when they are easier to fix. Also, as noted earlier, insight from the early stage model checking can lead to easier to verify designs, which can help the simulation reach desired state coverage goals.

### 4.1.2 Formal verification tool

We use probabilistic model checking with PRISM [47] to explore the design space of our DPM schemes and analyze trade-offs between efficiency, safety, and verifiability.

Using PRISM allows us to quantify the verification effort for the DPM scheme because the model checker traverses the entire reachable state space of a design, including the state transitions in ascertaining correctness. In recognition of the need for later detailed simulation of power management schemes, we use the number of reachable states and the number of transitions between states as metrics for the verifiability of a DPM scheme. The choice of probabilistic model checking over traditional, non-probabilistic model checking was motivated by characteristics of the
problem we want to verify. For the verification of a DPM scheme we are not only
interested in whether a power overshoot can happen, but also how often this is expected
to happen under typical conditions. These types of correctness characteristics depend on
the changing activity factor of the workloads, which can be captured in a probabilistic
framework.

The inputs to the probabilistic model checker are: the state elements of the system,
the probabilistic transition rules (a description of how the behavior can change from one
state to the next), and the correctness properties (the requirements which, if met, assure the
system’s correctness). In addition, it is possible to evaluate the expected values of certain
quantities in the system, such as power and performance, by associating rewards with
system states. Rewards are similar to tokens, in that the states that satisfy a certain
condition are assigned tokens. It is not our goal to use model checking for a better
estimate of power usage and performance impact; rather, we use the rewards to obtain
high-level measures of power and performance and analyze their trade-off with
verifiability. Based on the probabilistic state machine description, PRISM traverses the
entire reachable state space of the design and verifies whether the correctness properties
are met. PRISM also calculates the expected values of the performance and power
rewards over a certain bounded number of system transitions.
4.2  DPM Design Space Exploration

A wide variety of DPM solutions have been proposed in response to different requirements. In this section we describe the particular type of solution we analyze and its design parameters.

4.2.1 High Level View of DPM Design Space

We target DPM schemes that can cap the peak power usage of a multicore chip by using dynamic voltage and frequency scaling (DVFS). Figure 4-3 depicts the system we consider. The current state of each core is captured by its activity factor (measured as the core’s IPC), and by its voltage and frequency levels. The global DPM controller monitors the cores’ power use and periodically actuates their voltage and frequency levels. The overall goal of the global DPM controller is to maintain the power usage of the system below the budget target set by a user (which could be the OS) with a minimum performance penalty.

We use the expression “power budget” in a manner similar to prior work [40, 73]. The user desires that the power consumption of the core be maintained below this budget level (shown in Figure 4-4). The power budget differs from the Maximum Power for the chip, in that the budget is a somewhat soft limit. Exceeding the power budget occasionally can be tolerated if the amount of energy spent over budget is small. Budget overshoots cause the policy’s goal to be temporarily unmet, but they cause no thermal emergencies. Recently developed DPM schemes also allow temporary budget
overshoots [40, 73]. To keep the chip under its budget, the global controller periodically monitors the power usage of all cores and actuates their voltages and frequencies such that the total power consumption is maintained below the specified budget. We consider two actuation intervals: one for changing both voltage and frequency and one for changing only the frequency.

Figure 4-3: DPM scheme with global controller.

Figure 4-4 illustrates the power consumption of the chip over time. The Max Power horizontal line represents the maximum power the chip can consume given the worst case activity factors of all cores. The Budget line represents the constraint imposed on the power use of the chip. The global controller’s feedback mechanism uses this power budget value as the target to be reached.

In setting the voltage and frequency levels, the global controller makes the prediction that the cores will maintain their current activity factors for the next voltage actuation interval. When this prediction is not accurate, the actual power use can temporarily overshoot the power budget, as shown in Figure 4-4 at the times marked
with stars. On the next actuation point the controller tries again to bring the power use below budget.

![Figure 4-4: DPM scheme power utilization in time.](image)

### 4.2.2 DPM design goals and parameters

Of the multiple design goals that such a DPM scheme can target, we investigate efficiency (reducing the performance hit induced by decreasing core frequency through DVFS), safety (decreasing time and power spent over budget) and verifiability (decreasing required verification effort).

To reach these goals, designers can make decisions on many parameters. We consider here only a subset of them to keep our analysis tractable. Specifically, we compare a heterogeneous policy, which allows the controller to assign different voltage and frequencies across the cores, to a homogeneous policy, where the same voltage and frequency is set for all cores. For both policies, we analyze the design space along three parameters: number of voltage levels (VL) into which the voltage range is split, number of frequency levels (FL) that can be allocated for a given voltage level, and number of cores assigned to a single DPM controller.
Figure 4-5: Possible assignments of cores per controllers.

Figure 4-5 illustrates this “cores per controller” (CPC) design parameter. If we consider a 6-core chip, a DPM solution might use a single controller assigned to all cores (the outer boundary), or 2 controllers each monitoring 3 cores (the two horizontal groupings), or 3 controllers each supervising 2 cores (the three vertical groups).

4.2.3 Further motivating DPM early formal analysis

Designers certainly have some intuitive a priori understanding of how choosing different design points in the above parameter space affects their goals. For example, one might expect that for a multicore chip a heterogeneous solution with more CPC will outperform a solution with fewer CPC. This is expected to happen because with more CPC the power budget should be exceeded to a lesser degree due to averaging effects, making performance throttling less necessary. But what is the quantitative gain in performance when going from 2 CPC to 3 CPC, for example? Is that performance gain worth the impact on verification effort? How does the safety of the solution change in response to CPC? Do the answers vary between homogeneous and heterogeneous policies? In addition to questions about CPC, designers want to answer similar questions about other parameters, such as VL and FL, and possible interactions between
parameters. Will a change in VL impact design goals differently depending on the value of CPC?

These are the type of questions to which we seek answers via performing the proposed early stage formal analysis. These answers enable designers to make more informed decisions, and we show concrete examples of these benefits in Section 4.3.

4.2.4 Specification of correctness

There are three characteristics of the DPM scheme that we wish to verify. First, we want to verify that the DPM scheme is safe. A DPM scheme can be unsafe, for example, if it allows the power usage to often exceed the allocated budget. We consider two safety metrics that we use in our trade-off analysis: the percentage of time the system is expected to be over budget, and the percentage of power used over budget.

Second, we want to ensure that the scheme is correct, in the sense that it does not deadlock and it does not assign voltage and frequency values incorrectly. We verify that the following correctness properties are obeyed in every state of the system:

- No deadlock state can ever be reached;
- The voltages and frequencies for all cores are always maintained within a pre-specified range;
- There is no mismatch between the voltage and frequency assigned to a core (e.g., we never match a very high frequency with a very low voltage);
All DPM schemes that we evaluate in Section 4.3 meet the above correctness properties in every state, which is why this correctness factor does not appear in our trade-off analysis explicitly.

Third, we wish to verify that the DPM scheme is efficient in achieving as much performance as possible while not exceeding power constraints or violating priority rules for provisioning power. A buggy DPM scheme might sacrifice more performance than expected. Our efficiency metric is the performance loss due to the system’s functioning in reduced power mode compared to a baseline system with no DPM scheme.

4.2.5 DPM model

For our DPM scheme, the state elements are: the current voltage, frequency, and activity factor of each core and an incrementing counter triggering when the global controller should actuate both voltages and frequencies as opposed to only frequencies. Figure 4-6 shows a code snippet of the DPM description in PRISM’s language; the first part of the module defines the state elements.

The probabilistic transition rules specify how the activity factor changes for the cores and how the voltages and frequencies change in response to controller actuations. Figure 4-6 shows two example of transition rules. The first is a local transition (the local counter is smaller than the trigger value MAX_CNT) in which the frequency of the core can change. The second transition rule is a global one (the local counter reaches the
trigger value MAX_CNT) through which both the voltage and frequency of the core can change. We approximate each core’s activity factor using its instructions per cycle (IPC), because IPC is strongly correlated with the activity factor and it is easy to obtain. This correlation is not perfect, but obtaining the exact activity factor would require a low-level implementation that is unlikely to exist early in the design cycle. To make our analysis tractable with PRISM, we quantize the IPC values into four distinct ranges, and we choose the mean IPC of a range to represent the activity factor of a core in that range.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline width</td>
<td>4 decode/issue/commit</td>
</tr>
<tr>
<td>ROB/LSQ sizes</td>
<td>150 entries / 32 entries</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>2 level, 3 16K-entry BHTs</td>
</tr>
<tr>
<td>Functional units</td>
<td>4 FXU, 4 FPU, 1 BR</td>
</tr>
<tr>
<td>L1I cache</td>
<td>64KB, 2-way, 16B blocks, 1cycle</td>
</tr>
<tr>
<td>L1D cache</td>
<td>64KB, 2-way, 16B blocks, 1cycle</td>
</tr>
<tr>
<td>L2 cache</td>
<td>1MB, 8-way, 64B blocks, 9 cycles</td>
</tr>
<tr>
<td>Memory</td>
<td>100 cycles</td>
</tr>
</tbody>
</table>

Table 4-1: Microprocessor configuration.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Low Average IPC</th>
<th>High Average IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stable IPC</td>
<td>mcf</td>
<td>eon, crafty</td>
</tr>
<tr>
<td>Variable IPC</td>
<td>art, parser</td>
<td>bzip2</td>
</tr>
</tbody>
</table>

Table 4-2: Benchmarks.

We obtain the transition probabilities using Turandot [63], a detailed, cycle-accurate simulation model. The microprocessor’s configuration is shown in Table 4-1. For benchmarks, we chose six SPEC 2000 benchmarks, shown in Table 4-2, that have very different behavior, both in terms of their average activity factor and in how much
their activity factor changes over time. The appropriate SimPoint [74] intervals for these benchmarks were traced using Aria [64]. For each benchmark, the simulator produces the average IPC for each time quantum of 100μs (400,000 cycles at 4GHz). The sampling period of 100μs was chosen taking into consideration the longest duration of power spikes which could be allowed. Given that chip-level thermal time constants are in the range of milliseconds or tens of milliseconds [25], 100μs is a very safe, conservative setting for the sampling time.

We use rewards to keep track of power, performance, and the states in which the system is over budget. An example of how rewards are defined in PRISM can be seen in the last part of Figure 4-6. Below, we discuss in more detail the power and performance models we use. PRISM computes the expected rewards over a bounded interval, and we set the bound to 1000 system transitions (local actuation intervals) in our experiments.

In our performance model, the performance of a core is a linear function of its frequency, \( f \). That is, if we increase \( f \) by \( X\% \), then the performance is also improved by \( X\% \). This is an approximation, because the performance benefit of a large increase in \( f \) is limited by the unchanged memory performance. Nevertheless, for a high-level model that is considering small adjustments in \( f \), we think this assumption is reasonable. Our model considers the latency required to transition between voltage levels, and it assumes that a core functions at its lowest frequency during a voltage transition (1μs per 10mV). The latency of transitioning between frequency levels is much shorter—on the
order of one or two processor cycles [70]—because it can be done with on-chip digital PLL mechanisms. This latency is orders of magnitude shorter than a 100μs actuation interval, and thus we do not model it.

\begin{verbatim}
module Core1
    // State variables for Core1
    vddl: [VDD_MIN..VDD_MAX]; // current voltage level
    ipol: [IPC_L..IPC_H]; // current activity factor level (measured as IPC)
    fql: [0..FQ_LEV-1]; // current frequency level
    cnt1: [0..MAX_CNT-1]; // counter for triggering global controller actuation

    // Transition rules
    // This local transition is triggered when Core1 should be set to frequency level 2
    // according to DPM scheme, the counter is below threshold for global controller
    // activation, and Core1's activity level is 0; the transition sets the new frequency
    // to level 2, increments the counter, and updates the activity factor according to
    // predefined activity factor transition probabilities
    [local_trans] change_core1_freq_2 & cnt1<MAX_CNT-1 & ipol = 0 ->
    // with probability 0.85 remain at activity level 0
    0.85: (fql'=2) & (cnt1'=cnt1+1) & (ipol'=0) +
    0.10: (fql'=2) & (cnt1'=cnt1+1) & (ipol'=1) +
    0.05: (fql'=2) & (cnt1'=cnt1+1) & (ipol'=2);

    // This global transition is triggered when Core1 should be set to low voltage
    // level according to DPM scheme, the counter equals the threshold for global
    // controller activation, and Core1's activity level is 3; the transition sets the
    // voltage level to low, the new frequency to predefined level, resets the counter,
    // and updates the activity factor according to predefined activity factor transition
    // probabilities
    [global_trans] set_core1_VDD_L & cnt1=MAX_CNT-1 & ipol=3 ->
    0.3: (vddl'=VDD_L) & (fql'=2) & (cnt1'=0) & (ipol'=2) +
    0.7: (vddl'=VDD_L) & (fql'=2) & (cnt1'=0) & (ipol'=3);
Endmodule

// Reward structure to measure number of intervals spent over the power budget
rewards "Steps_over_budget"
    // Assign a reward of 1 to any state where the used power is greater than the budget
    used_power > power_budget : 1;
endrewards
\end{verbatim}

Figure 4-6: DPM code snippet in PRISM.
In our *power model*, the power consumption of a core is a function of the core’s frequency \( f \), voltage \( V \), and activity factor \( A \). We model both active and leakage power, with active power consumption formulated using the usual \(-f^a A V^2\) dependence equation. The leakage power is modeled approximately as a cubic function of \( V \), as this has been found to capture the behavior quite well for the particular supply and threshold voltage ranges appropriate for current CMOS technologies (65nm or 45nm). The power model used is admittedly abstract, but deemed to be good enough for the DVFS-driven power management policies considered in this work (as in Isci et al. [40] or Sharkey et al. [73]).

![Diagram of probabilistic model checking process]

*Figure 4-7: The probabilistic model checking process.*
4.2.6 Verification process

Based on the state variables and probabilistic transition rules, the model checker traverses the reachable state space of the system and evaluates whether the correctness properties are obeyed. If one of the state properties fails, a counterexample is generated. This process is illustrated in Figure 4-7. The cumulative value of the power and performance rewards is also calculated based on the power and performance models in the description of the FSM.

4.3 Case studies in verification of multi-core power management schemes

We now detail the two specific DPM schemes we modeled for our analysis and their design parameters. Then we describe the performance, safety, and verifiability trade-offs we find in this design space.

4.3.1 Scope of analysis

We analyze heterogeneous and homogeneous DPM schemes. For the heterogeneous schemes, the controller uses a priority based greedy algorithm for distributing the power budget. It allocates the largest voltage that fits in the power budget for the first core (while provisioning enough power to run the rest of the cores at lowest voltage) then allocates the largest possible voltage for the second core and so on. This heterogeneous policy is very similar to current state-of-the-art DVFS policies, such as the “Priority” scheme analyzed by Isci et al. [40]. For homogeneous schemes, the controller allocates the single greatest voltage level that keeps the chip below the power budget, assuming
all cores maintain their current activity factors. This homogeneous policy is very similar to the “Chip-Wide DVFS” scheme proposed by Isci et al. [40].

All of our DPM schemes use two actuation intervals: a 500μs one to change both voltage and frequency of cores (the frequency is set to the highest value permitted for the voltage level selected) and a 100μs one to change only the frequency. We vary the voltage range from 1.05V to 0.78V and we scale the frequencies linearly with the voltage from 4.2GHz to 3.15GHz.

When analyzing the impact of increasing VL, we maintain the same voltage range and divide it into more levels (from 2 to 6 in our experiments). When varying FL, we divide the frequency range corresponding to a particular voltage level into more values (from 1 to 5). We also vary CPC from 1 to 3. Note that this is different from comparing a 1-core chip to a chip with 2 or 3 cores; we consider a chip with the same number of cores, 6 for example, which has 6, 3 or 2 controllers. We do not model a 6-core system with a single controller (having CPC of 6) because the associated state explosion makes the verification through model checking impractical and our results show little overall performance improvement beyond 3 CPC.

In our analysis, the global controller uses the power model described in Section 4.2.5 to estimate the power use of the system (a function of activity factor, voltage and frequency). The global controller predicts that the cores will maintain their current activity factor during the next interval.
We perform a range of experiments setting the power budget to 25, 40, 50, 70 and 100% of the maximum power the chip can consume (corresponding to a 4 IPC activity factor across all cores). The results we present are averaged across the different budget levels and benchmarks.

### 4.3.2 Impact of number of voltage levels

The first design parameter we explore is VL. We consider a heterogeneous scheme and fix FL to 2 for clarity (the results were similar for the other FL values). Figure 4-8(a) shows the impact of VL on performance with respect to a chip without DPM. Figure 4-8(b,c) show safety, and Figure 4-8(d,e) show verifiability.

We notice a strong interaction between VL and CPC; on many of our metrics of interest, the impact of increasing VL varied across different levels of CPC. Hence we present data for CPC=1, 2 and 3 on the same graph.

We note several interesting phenomena. First, in terms of performance, the trend corroborates our intuition that increasing VL benefits performance. However, we notice a saturation around VL=5 and performance remains almost flat afterwards. Prior work [73] proposed using VL=10 in an experimental setup that used 4 cores, simulating various SPLASH benchmarks. Our results, albeit in a different setup, suggest that such a large value of VL offers little marginal benefit.

In terms of safety, the percentage power spent over budget is minimal, ranging from 0.1% to < 0.5% of the power usage of a solution without DVFS. The percentage of
intervals spent over budget varies from ~0.5% to ~9%. An increase in CPC allows the controller to make more aggressive decisions in matching the power budget resulting in more mispredictions. The same can be said about increasing VL. Whether the amount of time spent over budget is deemed tolerable or not depends on the particular constraints of the application. However, considering the tiny percentage of power spent over budget, we conclude that VL does not greatly impact safety.

Figure 4-8: Impact of Number of voltage levels (VL).

Given only the performance and safety analysis of the design space, one might conclude that the greatest difference can be noticed when going from CPC=1 to CPC=2
and that there is a minimal difference between CPC=2 and CPC=3. However, if we add verifiability to the picture, the conclusion changes dramatically. The verification effort, measured both in number of reachable states and transitions, increases dramatically with CPC. We see a strong interaction between CPC and VL in terms of verifiability effects. For both the CPC=1 and CPC=2 solutions, the verification effort does not increase significantly with VL, unlike the case for the CPC=3 solution.

In conclusion, the performance improvement gained from going from CPC=2 to CPC=3 is insignificant (particularly for larger VL) while the increase in verification effort is extremely large. Our data suggest that the better design solution consists of having multiple controllers each assigned to a small number of cores (2) which can be set to 4-5 voltage levels as opposed to a design with a large CPC at low VL.

### 4.3.3 Impact of number of frequency levels

The second design parameter we address is FL, the number of frequency levels that can be set for a given voltage level. Our hypothesis was that the 100μs actuation of the controller can take advantage of the increased frequency granularity and better track the power budget between consecutive voltage actuations.

Figure 4-9 shows our results when we consider a heterogeneous policy and fix VL=3 for performance with respect to a chip without DPM (a), safety (b, c) and verifiability (d, e).
Our results indeed show a slight improvement in safety due to the increased flexibility in frequency levels. However, this improvement is minimal and accomplished with a performance penalty. The reason is that the frequency decrease is a lot less efficient in decreasing the overall power usage than the voltage. The impact of FL on verification, however, is very large both in reachable states and transitions. We conclude that the frequency knob should be used only when the safety margins of being over budget are tight, because a significant cost in verifiability will be paid. Also, FL=2 seems
to suffice for getting most of the safety benefit. Our conclusion is specific to the type of system we analyzed, where it is possible to set both voltage and frequency of individual cores at different levels. For this case, using many frequency levels for one voltage level does not seem to represent a good design alternative from a verifiability, performance, and safety trade-off. For the class of systems that allocate the same voltage across all cores, the impact of frequency levels is likely to be more beneficial.

4.3.4 Using a homogeneous versus a heterogeneous scheme

We now explore the impact of choosing a homogeneous policy. We wish to discover whether homogeneity helps or hurts our pursuit of better design points. Figure 4-10 shows the results for a homogeneous policy when we vary VL. We notice a slight decrease in performance for an increase in CPC. This result is due to the fact that the homogeneous policy is more restrictive and all cores assigned to the controller are throttled to a single voltage level to match the budget. Second, the performance impact of increasing VL is more significant compared to the heterogeneous case. The safety is improved for the homogeneous solution as the percentage of intervals spent over budget decreases significantly.
4.4 Guidelines for verifiability

With insight from the case studies discussed above, we next propose two higher-level guidelines for promoting the design of verification-aware DPM schemes.

**Guideline 1.** For limiting the verification effort without losing much efficiency, try to divide the chips into small clusters of cores, where each cluster would use a global control scheme as opposed to using a single global controller for all cores.

**Guideline 2.** For systems where both voltage and frequency can be set individually for each core, assign a small number of frequency settings for each voltage level, as high granularity at frequency level increases verification effort to a large extent without a significant improvement in efficiency and safety for a DPM scheme.
4.5 Related work

Power management is an important issue and thus there has been a significant amount of prior work in this area. In this section we first present multicore-specific power management schemes (Section 4.5.1) and then discuss prior work in power management verification (Section 4.5.2).

4.5.1 Multicore power management

The most straightforward way to manage power in a multicore chip is to simply apply well-known single-core techniques to every core. However, Isci et al. [40] observed that such “local” (per-core) management was potentially inefficient because it could not take advantage of peak power averaging effects that occur across multiple cores. They introduce global schemes in which a single, centralized, “global” controller determines the power budget and settings (e.g., voltage and frequency) for every core. Sharkey et al. [73] provide a more detailed evaluation of these global schemes in terms of their efficiency. Sartori and Kumar [72] present a proactive scheme for managing peak power in multicore chips. They observe that distributed algorithms can be used to select the power level allocation for cores and that they would be more scalable than algorithms based on having a centralized global controller. However, no multicore DPM scheme has been analyzed to determine its verification effort and to trade-off verifiability against other design goals.
4.5.2 Verifying power management schemes

There has been a limited amount of prior work in verifying DPM schemes. One representative piece of work by Shukla and Gupta [75] uses the SMV model checker [56] to verify a DPM scheme. We are interested in DPM for multicores, whereas their focus is on solutions for unicro systems. Furthermore, we use model checking to estimate verification effort and verify a set of correctness properties, while they use it to stress the optimality bounds of the DPM scheme by constructing a worst case task trace. Dubost et al. [30] present a high-level argument for specifying power management schemes in the Esterel language, which facilitates using a model checker to verify the designs. They do not discuss any specific DPM scheme or verification.

Two research papers [48, 65] have used the PRISM probabilistic model checker [47] to analyze DPM schemes. They target unicro systems and use PRISM to find optimal power management policies for given task arrival distributions and constraints on expected wait queue size. In contrast, we are interested in analyzing the trade-off between verifiability and other metrics for multicro schemes.

4.6 Conclusion

Power management is important for multicro processors, and DPM scheme designers would like to have confidence that their schemes are both safe and efficient. We have shown the insight that can be gained by using formal methods—in this case, probabilistic model checking—to analyze high-level descriptions of DPM schemes. We
have used PRISM to determine the effort required for verifying DPM schemes, and we have compared these schemes with respect to their efficiency.

One conclusion we draw from this work is that global schemes (i.e., CPC>1) offer significant benefits in performance due to the ability to balance power across more cores. However, we must be careful to avoid scaling them to more cores than necessary. Linear increases in CPC cause exponential increases in the size of the reachable state space. Thus it is important to find the system configuration where both the verification is tractable and we obtain the majority of the benefits of a global solution. Our data shows that much of the benefit is achieved at just CPC=2; increasing CPC further provides little additional performance gain. In terms of safety, we found no significant difference between percentage energy spent over budget as a function of CPC, but a larger value of CPC resulted in the system spending more time over budget. Thus we recommend designs in which chips are divided into small clusters of cores, where each cluster uses a global control scheme.

A second conclusion is that the use of fine-grained frequency tuning is likely not worth its costs for systems where it is possible to set both voltage and frequency of individual cores at different levels. The results show that having a large FL has an extremely large impact on verification effort. It is not clear that its modest safety benefits justify these verification costs.
5. Verification-aware design for unicore power management schemes

Maintaining a low energy envelope remains one of the critical requirements in current processor designs. Until several process generations ago, the active energy consumed by transistors switching on or off dominated the total energy consumption. However, every new process generation brings a decrease in key process parameters such as oxide thickness, gate length, and threshold voltage, and these feature reductions lead to an exponential increase in leakage (static) energy. Because of this trend, the proportion of total chip energy represented by the leakage component has increased dramatically. The semiconductor industry’s 2007 roadmap identifies energy consumption as an “urgent challenge” and its leakage component as a “major industry crisis in the long term, threatening the survival of CMOS technology itself”[1].

Many mechanisms have been proposed for reducing leakage energy, including circuit level and microarchitectural schemes. One of the most promising mechanisms is power gating, whereby leakage energy is saved by cutting the supply voltage to idle circuits. Recently, Hu et. al [36] demonstrated how to effectively use microarchitectural information to predict when to power gate a microprocessor’s units, such as its functional units. Over long time scales (i.e., the runtime of a benchmark), they show that power gating often saves a significant amount of leakage energy at a small performance cost due to waiting for units to wake-up from power gating.
Power gating can potentially save a significant amount of leakage energy but, because it is predictive, it can also lead to significant energy penalties when mispredictions are frequent. The desired behavior (correctness specification) of a power gating scheme is for the system to save leakage energy in comparison to a system without power gating. However, this desired behavior is not guaranteed. Every time a component is power gated off, a certain amount of energy is necessary for turning the component off and then on again. If the power gated unit remains idle for a long interval, the wake-up energy cost is more than offset by the leakage energy saved by power gating during the idle time. If, however, the power gated unit needs to be brought back in use soon after being turned off, that energy cost is not offset. In this case, the power gating feature incurs an energy penalty with respect to a unit without power gating.

We show in this work that the impact of power gating is quite dependent on the workload running on the processor, and we observe that both big wins and big losses can occur. Across the SPEC2006 benchmarks, we observe a large variability in the leakage energy savings of a well-known power gating algorithm [36] applied to functional units. On a given benchmark, applying the power gating algorithm for a functional unit on certain time intervals can save as much as 99% of its leakage energy, whereas during other time frames it can consume 70% more energy than a system without power gating. We also see large variability in energy savings between benchmarks. For the same functional unit, some benchmarks can save as much as 88% of their leakage energy
while others may consume 17% more energy than without power gating. This wide gap between a large potential energy savings and a large potential energy penalty decreases the confidence in the value of adding power gating to the processor.

When adding a feature such as power gating to a processor, we would like for it to both benefit the common case and do no harm. To assess the common case benefit (e.g., average energy savings), it is generally sufficient to simulate the system across a wide range of benchmarks. To assess the potential for harm (e.g., consuming 70% extra energy), this simulation approach is often insufficient, because a harmful situation may be unacceptable even if it is unlikely. Consider for example the case of a power virus application designed specifically to attack the system by taking advantage of this vulnerability and increasing its energy consumption. When deciding whether to add a feature to a real design, we must take into account the feature’s behavior for applications that have not been simulated. Such an unsimulated behavior could be, for example, a scientific application running in a loop that consistently wastes a significant amount of energy through the addition of the power gating scheme. This consistent waste of energy is a design vulnerability of the power gating scheme that might prevent its adoption in a real processor design.

When a proposed feature such as power gating can have both a large benefit and a large penalty, we would like to augment it with two mechanisms. First, we want a mechanism that enables the feature when that is likely to translate into a win and
disables it when the result is likely to be a loss. Second, we want a mechanism that provides a \textit{guaranteed bound on the worst-case behavior} of the feature, in order to increase our confidence in the feature’s value and improve its likelihood of adoption. In this work, we propose the addition of both of these mechanisms in the context of power gating. Specifically, our contributions are the following:

- We propose augmenting current dynamic power gating schemes with a Success Monitor that can dynamically estimate whether a particular power gating scheme is successful at saving energy or not.

- Based on the Success Monitor, we implement a feedback control mechanism called the Success Monitor Switch that senses the energy savings of the power gating scheme and can enable/disable it depending on its success. This switch does not increase, on average, the total energy savings. However, we show that for the benchmarks initially exhibiting energy loss, the Success Monitor Switch saves energy. It also decreases the performance penalty, on average, by 71% across the SPEC2006 benchmarks.

- We implement a Token Counting Guard mechanism that can provably bound the worst-case energy penalty of the power gating scheme to an acceptable threshold (of 2% of leakage energy in our experiments) over a specified interval of time (100 monitoring intervals in our experiments). With the Token Counting Guard in place, the power gating scheme saves, on average, marginally more energy (with 3.5%),
and its performance impact is reduced by 52%. The largest benefit of the Token Counting Guard though, does not come from the marginal increase in energy savings but from its ability to bound energy loss for any runtime behavior.

- We evaluate the effect of using the Success Monitor Switch in combination with the Token Counting Guard. This mixed solution retains the benefit of bounding the energy loss while slightly increasing the average energy savings (by 2%) and decreasing the performance penalty by 63%.

- We demonstrate that functional correctness at microarchitectural level can be ensured for a system implementing power gating of processor functional units, with or without our mechanisms, with low extra verification effort.

5.1 Power gating potential and pitfalls

In this section we discuss some background on how power gating works, its quantitative potential for leakage energy savings, and its possible pitfalls.

5.1.1 Background

Power gating requires, for each circuit that can be turned off, the presence of a header (or footer) “sleep” transistor that can set the supply voltage of the circuit to ground level (or VDD level for footer) during idle times. Figure 5-1 illustrates power gating using a header transistor. Power gating also requires control logic to predict when would be a good time to power gate the circuit.
Figure 5-1: Example of a power gateable circuit using a header transistor.

Every time the control logic decides to power gate the circuit, an energy overhead cost is incurred. This energy overhead is due to 1) distributing the sleep signal to the header transistor before the circuit is actually turned off and 2) turning off the sleep signal and driving the Virtual Vdd when the circuit is powered-on again. The energy savings resulting from power gating is a non-linear function of time. Figure 5-2 depicts a schematic of the cumulative energy savings achieved during power gating. On the left side of the figure, the control algorithm decides to power gate the circuit and on the right the unit is turned on again. For more details on how such a curve can be derived, the interested reader is referred to Hu et al. [36]. The break-even point on the figure represents the point in time where the cumulative leakage energy savings equals to the energy overhead incurred by power gating. If, after the decision to power gate a unit, the unit stays idle for a time interval that is longer than the break-even point, then power gating saves energy. If, however, the unit needs to be active again before the break-even point is reached, then power gating incurs an energy penalty. Adopting the terminology used in by Hu et al. [36], we call the time between power gating and when
the unit has reached the break-even point “uncompensated,” and we call the time after the break-even point “compensated.”

Figure 5-2: The break-even point.

Figure 5-3 shows two example functional unit utilization patterns for a break-even point equal to nine cycles. A “zero” in the pattern signifies that the unit has been idle for that cycle while a “one” means that the unit was utilized. The arrow indicates the point at which the decision is made to power gate the circuit. For the upper utilization pattern, the prediction to power gate the unit ends up saving energy because the unit remains idle for longer than the break-even point of 9. However, the lower pattern represents a misprediction for the control algorithm because the unit needs to become active again the third cycle after being turned off, resulting in an overall energy penalty.
In terms of possible performance impact of power gating, this impact can come from two different sources: the addition of the “sleep” transistor and supplementary cycles spent waiting for functional units to wake up. The addition of the “sleep” transistor can imply a small hit on cycle time. This can be avoided by a small increase in VDD level to counteract the voltage droop across the “sleep” transistor. The second source of performance impact (additional cycles spent waiting for unit wake up) can be mitigated to a large extent by using information from the decode stage or issue queue to proactively wake up needed units [33].

5.1.2 An example power gating scheme

The baseline power gating scheme that we will augment in this work is a microarchitectural technique for power gating functional units that was developed by Hu et al. [36]. They evaluate the potential for power gating the functional units and propose an IdleCount algorithm for deciding when to assert the power gate signal. The IdleCount algorithm counts the number of cycles a unit has been idle and decides to shut the unit off when a fixed threshold (the idle_detect) has been reached. They also
propose another algorithm that triggers power gating after a branch misprediction. We will illustrate two of our proposed mechanisms, the Success Monitor Switch and Token Counting Guard, by adding them on top of the IdleCount algorithm. However, as we discuss in Sections 5.3 and 5.4, our mechanisms are not restricted to the IdleCount algorithm.

### 5.1.3 Power gating potential

Since implementing power gating in a system has associated costs in terms of voltage noise, performance penalty, area and power overheads [43], workloads must have a significant amount of idleness for power gating to be effective. In this section we show that this is indeed the case by exploring the power gating potential present for functional units in the SPEC 2006 benchmarks.

The data presented was obtained by following the methodology and energy model presented in detail in Section 5.6. We use bit-vector utilization traces from a very detailed, cycle-accurate PowerPC™, simulator representing a state-of-the-art superscalar microprocessor. The processor model includes 2 floating point units (FPUs), 2 fixed point units (FXUs), and 2 load-store units (LSUs). Details of the modeled microarchitecture are described in Section 5.6.1.

For the integer benchmarks, Figure 5-4 (left) shows the energy saving potential of each unit, as a percentage of the total leakage energy of that unit, as a function of the break-even point. Considering the analytical model developed by Hu et al. [36], the
break-even point for functional units developed in current technology parameters is in the range of 9 to 24. The energy savings values are calculated for an oracle algorithm that has knowledge of the future behavior and turns a unit off immediately when that decision is a correct prediction that saves energy (i.e. an idle period follows that is longer than the break-even point). The oracle also wakes-up a power gated unit when required without incurring any performance penalty at wake-up.

![Figure 5-4: Best-case potential to power gate for the SPECint (left) and SPECfp (right) benchmarks.](image)

Overall, we observe a large potential for power gating across all units, with the lowest being for the FXU units: 20%-35% for FXU0, 25%-50% for FXU1, 20%-40% for LSU0, 50%-65% for LSU1, and almost 100% for both FPU units due to their very low utilization in the SPEC 2006 integer benchmarks. Furthermore, we see an additional interesting trend. There is a significant increase in power gating potential associated with a decrease in the break-even point. This result is intuitive since a lower break-even point signifies that more idle intervals can be effectively used for power gating by the oracle algorithm. The implication from this data is that if circuit-level techniques become
available to reduce the energy overhead associated with power gating, then the additional energy savings are significant.

Figure 5-4 (right) shows similar data for the FP benchmarks. We observe similar trends with the exception of the FPU units being utilized to a higher degree in the FP benchmarks, translating into an energy savings potential in the range of 40%-60% for FPU0 and 60%-70% for FPU1. The high potential for saving leakage energy by power gating processor units justifies the application of microarchitectural level techniques for harvesting it. However, power gating can have its own pitfalls which we present in the next section.

### 5.1.4 Power gating pitfalls

The high level of idleness of functional units present in the SPEC 2006 benchmarks makes power gating at functional unit granularity an attractive solution for decreasing the overall processor leakage energy. However, in practice, before deciding whether to implement such a solution in a real processor, designers require a high degree of confidence that the proposed scheme is robust in achieving its desired goals. For power gating, the goal is to save energy or at least not to waste a significant amount of energy when enabled in a processor. In this section, we show that this goal is not always achieved by current power gating schemes. Knowledge of a large window of future behavior of a processor unit is necessary to decide whether power gating that unit will translate into energy savings or an energy penalty. Due to the inherently speculative
characteristic of power gating algorithms, it is possible for power gating schemes to result in significant energy penalties when enabled. We argue that this possibility represents a vulnerability of the power gating algorithm that could result in a design decision against implementing the scheme in a current real processor design.

To investigate this possibility, we implemented the IdleCount algorithm proposed by Hu et al. [36]. In the rest of this work, we will use the IdleCount scheme as a baseline to which we will add our mechanisms and to which we will compare their effectiveness. IdleCount counts the number of successive cycles when a particular unit is idle. When that counter reaches a predefined threshold value (called idle_detect), the decision is made to power gate the unit. Upon the unit becoming active again, the counter is reset. We ran the IdleCount algorithm on the SPEC2006 traces and computed the energy savings using the energy model presented in Section 5.6.2. Several of the SPEC2006 benchmarks (dealII, leslie3d, gobmk, and hmmer) exhibited an energy penalty due to the addition of the IdleCount power gating feature compared to an initial system without power gating. We also tested a microbenchmark called fldaxpy that exhibits a type of behavior that is difficult for the speculative power gating scheme to predict correctly. Figure 5-5 presents the energy penalty due to power gating across all these benchmarks in the FXU0 unit for a value of the idle_detect threshold of 5. The solid bars represent energy savings assuming zero performance penalty at wake-up from power gating while the striped bars assume a three cycle performance cost at wake-up. We see
that it is indeed the case that the speculative power gating algorithm can incur significant energy penalties.

![Figure 5-5: Example benchmarks wasting energy with the IdleCount scheme.](image)

We conclude that a feature added to save energy can at runtime end up costing extra energy. Moreover this type of runtime behavior can be encountered in regular benchmarks. To reduce the worst-case impact of power gating (and predictive schemes, in general), we propose the addition of two mechanisms to the policy, which we discuss in the next section.

### 5.2 Specification of power-gating scheme correctness

We consider two aspects of correctness that must be verified when adding a power gating scheme to a CMP system. The first aspect relates to the extent to which the power gating scheme reaches its goal of saving leakage energy at runtime. The second correctness specification is to ensure that the system remains functionally correct after the addition of the power management mechanism.
5.2.1 Power savings correctness specification

As was discussed in the previous section, power gating is a predictive technique that might either save energy or waste energy depending on runtime application behavior. Without any guard mechanism in place, it is possible for a power gating solution to end up consuming a significant amount of energy instead of leading to any energy savings.

One possible specification could require that a correct power gating scheme should always save energy, or in other words that the power gating scheme never wastes more energy than it saves. However, we notice that enforcing such a correctness requirement is excessively restrictive for predictive techniques such as power gating. Unless an oracle exists for future activity patterns of power gateable components, power gating mispredictions cannot be completely avoided. In other words, if wasting any amount of energy through the use of power management is not acceptable, the only option for a predictive power gating scheme to meet such a requirement is to be disabled. This is true because unless the predictor triggering power gating is 100% accurate, it will always be possible for a unit to be needed immediately after power gating, leading to an energy loss.

We propose the use of another correctness specification that is less restrictive, but at the same time provides a strong quality guarantee to designers. Our proposed correctness specification sets a non-zero bound on the worst-case possible energy
penalty associated with mispredictions over a certain time interval. The correctness specification then reads “In the worst case, the power gating scheme will not lose more than X% (X>0) of its leakage energy over a time slot of Y actuation intervals”. The mechanisms we describe in Section 5.4 ensure that this correctness specification is met and we thus describe them as “design for verification” examples.

5.2.2 Functional correctness specification

The second correctness criteria to be verified after the addition of a power gating scheme is that of preserving functional correctness. For example, a processor with power gateable units must coordinate the power-gating and wake-up of functional units such that the processor generates correct results. In Section 5.5 we discuss the implication for functional correctness of making functional units power-gateable.

5.3 Proposed mechanisms to improve the common case

We propose adding a Success Monitor to predictive mechanisms (e.g., power gating) whose success or loss depends on the dynamic behavior of the application. The goal of the Success Monitor is to estimate whether a particular policy is successful or harmful during a certain time interval that we call a monitoring interval. Based on the estimate from the Success Monitor, the control logic can better predict when to power gate (discussed in Section 5.3.2).
5.3.1 Success Monitor Switch structure and behavior

The Success Monitor requires four values to dynamically estimate the success or loss of a policy, two of which are obtained at runtime and two of which are static:

- The number of successful instances over the monitoring interval (runtime),
- The number of harmful instances over the monitoring interval (runtime),
- Reward of a successful instance (can be approximated to a constant for a particular system), and
- Cost of a harmful instance (can be approximated to a constant for a particular system).

We refer to the counters that are used to store the runtime values as efficiency counters.

In the context of power gating, a successful instance is any compensated cycle (i.e., a cycle when a power gated unit remains idle after reaching the break-even point). We keep track of energy savings or penalties by using tokens, one token corresponding to the leakage energy used by the unit during one cycle. The reward of a successful instance is thus one token. A harmful instance is represented by any case when the unit needs to be woken up before reaching the break-even point. We pessimistically assign a cost equal to the Energy Overhead for that unit (refer to Figure 5-2) for any harmful instance. The Energy Overhead value represents a pessimistic estimate because the unit might in fact have been idle for a significant number of cycles before being woken up.
(and thus could have recouped some of the Energy Overhead). We have experimented with two other estimates for the cost of a harmful state: a) an optimistic estimate that assumes the unit has been idle until just before the break-even point and b) an expected value estimate that is calculated by giving equal likelihood to the unit returning to the active state from any cycle between 0 and the break-even point. Experiments with these alternative estimates have resulted in similar results to the experiments using the pessimistic estimate. Using the pessimistic estimate has the advantage that it allows us to guarantee, by using the Token Counting Guard described in Section 5.4.1, that the energy penalty is below the bound set by the user.

5.3.2 Using the Success Monitor Switch to improve prediction

This information from the Success Monitor can be used by a hardware mechanism or by a high level software entity to dynamically change the power gating policy. In this work, we use the Success Monitor to drive an enable/disable signal for the power gating control logic. When the Success Monitor estimates that the power gating policy has been harmful over the previous monitoring interval, we disable the policy during the next monitoring interval. If the Success Monitor estimates that power gating was successful, then the policy is maintained enabled. We call this solution the Success Monitor Switch. The efficiency counters are incremented and made available to the Success Monitor regardless of whether the power gating policy is enabled. This permits
the Success Monitor Switch to turn power gating on again when the application has reached an interval in which power gating is expected to be beneficial.

Figure 5-6: Efficiency counters.

Figure 5-6 shows a schematic of the monitoring mechanism and this particular scenario. The lower part of the figure depicts the baseline IdleCount algorithm under which the system can be in one of three states: on (or active), power gated off but uncompensated (Off_U in the figure), and power gated off and compensated (Off_C). Two efficiency counters are incremented depending on the dynamic application behavior. The counter for successful instances is incremented each time the unit remains in a power gated compensated state (Win++ in the figure). The counter for harmful instances is incremented each time the unit goes from a power gated uncompensated state to being active again (Lose++). This situation is harmful because the unit has not been idle for a sufficiently long interval after being power gated to compensate for the
Energy Overhead. The upper part of Figure 5-6 shows the monitor and control level of the power gating scheme which keeps the lower level scheme enabled if it estimates that the scheme will save power.

5.3.3 Added hardware for the Success Monitor Switch

Three counters are required for the Success Monitor Switch: the power gating counter, the success efficiency counter and the harmful efficiency counter. The power gating counter keeps track of the number of cycles a unit is idle after being power gated, up to the break-even point. The success efficiency counter is incremented each time the power gating counter is at break-even point value and the unit remains power gated. The harmful efficiency counter is incremented whenever the power gating counter is less than the break-even point and the unit needs to wake-up from a power gated state. Both efficiency counters are reset at the beginning of each monitoring interval. Table 5-1 details the number of latches necessary for each of the counters. Calculating the total cost of all harmful instances in $E_{L,cyc}$ units (leakage energy for that unit per cycle) can be done by performing a shift on the harmful efficiency counter. For example, if the cost of a harmful instance is $8 \ E_{L,cyc}$, a 3-bit left shift on the harmful counter will calculate the total cost. To calculate whether the scheme is successful or not, a 6-bit subtractor is necessary that takes as inputs the success efficiency counter and the shifted value of the harmful efficiency counter. Two incrementers are also required for updating the values of the efficiency counters.
Table 5-1: Latches necessary for the Success Monitor Switch.

<table>
<thead>
<tr>
<th>Added Hardware Component</th>
<th>Formula for computing number of latches</th>
<th>Number of latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>power gating counter</td>
<td>$\lceil \log(\text{break_even_point}) \rceil$</td>
<td>$\lceil \log(19) \rceil = 5$</td>
</tr>
<tr>
<td>success efficiency counter</td>
<td>$\lceil \log(\text{monitoring_interval}) \rceil$</td>
<td>$\lceil \log(50) \rceil = 6$</td>
</tr>
<tr>
<td>harmful efficiency counter</td>
<td>$\lceil \log(\text{monitoring_interval}(\text{idle_detect} + 1)) \rceil$</td>
<td>$\lceil \log(50/(5 + 1)) \rceil = 4$</td>
</tr>
</tbody>
</table>

5.4 **Design for verification of power-gating schemes**

Our second proposed mechanism is the Token Counting Guard that provides a provable worst-case bound on the possible penalty associated with mispredictions. Having an explicit mechanism to ensure this bound on the worst case behavior of a power gating scheme illustrates a “design for verification” approach in the context of power gating. The presence of a worst case behavior guard mechanism separates the issues of average efficiency of a scheme from its correctness (as defined in Section 5.2.1). The average efficiency of the power gating scheme can be checked through simulation of common application behavior as found in the SPEC2006 benchmarks for example. However, validating the power gating scheme through simulation has difficulties in identifying incorrect infrequent behavior. It is these rare, incorrect behavior patterns that are caught by our proposed Token Counting Guard mechanism described below. As with the Success Monitor mechanism, we discuss the Token Counting Guard in general but describe details for its specific application to power gating.
5.4.1 Bounding the worst case with a Token Counting Guard

The proposed Token Counting Guard mechanism can provide a guarantee on the worst-case behavior of a policy. The guarantee is given over a time interval, called the *guarantee interval*, which is an integer multiple of the monitoring interval (Figure 5-7a).

![Diagram](image)

**Figure 5-7: The token bag mechanism.**

We associate tokens with the quantity we wish to limit. For example, in the case of power gating, we equate one token to the leakage power of the unit over one cycle. A token bag holds the tokens that a unit can consume over the course of one guarantee interval, as illustrated in Figure 5-7b. Figure 5-7c shows how the number of tokens in the token bag varies over time. The value of the token bag is updated as follows. At the beginning of a guarantee interval, the token bag is reset to a fixed, non-zero value that represents the entire amount of energy penalty that can be tolerated over the current guarantee interval. For example if we wish to guarantee a maximum leakage energy
penalty of 2% over 100 monitoring intervals each 50 cycles long, then the token bag is initiated to 100 tokens\(^1\).

At the end of each monitoring interval, the token bag is updated depending on the energy savings or penalty estimated by the Success Monitor over this interval. The token bag will be increased if energy was saved or it will be decreased if energy was wasted. The quantity by which the token bag is updated corresponds to the token equivalent of the energy saved or wasted.

At the beginning of each monitoring interval, a decision is made, based on the number of tokens remaining in the bag, whether to enable the feature (e.g., power gating) for the next monitoring interval or not. If there are enough tokens left to tolerate the worst possible behavior of the feature for the next monitoring interval, then the feature is enabled. Otherwise the feature is disabled. Once the feature is disabled due to insufficient tokens to cover the worst-case energy penalty, it is no longer enabled until the end of the guarantee interval when tokens become available again. The benefit of the token bound mechanism is that it limits the penalty incurred by a feature in the worst-case scenario. However, we wish to achieve this bound without being overly restrictive (i.e. without disabling power gating when it could save energy). The key to achieving this goal is that there is a significant amount of energy savings slack across one guarantee interval for most workloads. The power gating scheme is only disabled when all the tokens have been

\(^1\) 100 intervals * 50 cycles/interval * 2tokens/100 cycles = 100 tokens
consumed for that guarantee interval. When this happens, it is likely that applying the power gating scheme would have continued to result in an energy penalty until the end of the guarantee interval. By disabling power gating only in instances when it probably wastes energy, we see slightly higher energy savings for a system with Token Counting Guard compared to a system without it.

The application of the Success Monitor and of the Token Counting Guard is not restricted to power gating schemes nor to power management in general. Any feature that, depending on runtime behavior, can succeed or not in reaching its goals can benefit from these mechanisms.

### 5.4.2 Added hardware for the Token Counting Guard

In addition to the power gating and efficiency counters described for the Success Monitor Switch, a 12-bit register is necessary for holding the value of the token bag\(^2\). This register is initiated to a fixed value at the beginning of each guarantee interval (100 in our experiments for a 2% bound over 100 monitoring intervals). A 12-bit adder is also necessary for updating the value of the token bag at the end of each monitoring interval.

---

\(^2\) When the scheme is saving energy, the token bag can accumulate tokens above the initial 100 tokens. During one monitoring interval a maximum of 50 tokens can be saved and a maximum of 100 consumed. The maximum value the token bag needs to hold is in fact 3400 (less than \(50 \times 100\)). It can be calculated from the limit case when until the monitoring interval \(i\) the scheme has maximum energy savings and after interval \(i\) the scheme has maximum energy penalty: \(100 + 50i = 100(100-i)\) has \(i = 66\) as solution leading to a maximum value of 3400 for the token bag. In any case other than this limit case the token bag needs to hold a value lower than 3400.
Individual power gating, efficiency counters and token bag registers are required for each power gateable unit. However, the rest of the hardware necessary for the Success Monitor Switch and Token Counting Guard can be shared across all power gateable units. They can be multiplexed across the units because the monitoring of the units can be done at different cycles for different units. We did a careful analysis of the added hardware overhead, by calculating the "latch count equivalent" of all the components within the Success Monitor Switch and the Token Counting Guard. Based on known VHDL-derived latch counts, and macro-level area estimates of the processor core, we were able to (very conservatively) bound the area and power overhead of the added hardware to at most 0.06 % of the full core.

5.5 Implications for functional correctness and verification

In this section we discuss our expectations for the verification consequences of both power gating (Section 5.5.1) and our proposed mechanisms on verifying the processor’s functional correctness (Section 5.5.2). Our discussion is restricted to implications on design verification, at the microarchitectural level, where we pursue our work. Thus we do not refer to circuit-level verification techniques and changes due to power gating that are not visible at the microarchitectural level. We assume that the initial processor design without power gateable units is correct.
Figure 5-8: Interface between the power gating scheme and the rest of the system. The shaded block illustrates our proposed mechanisms: the Success Monitor Switch and the Token Counting Guard.

In both discussions, we refer to Figure 5-8, which shows the interface between the functional unit that can be power gated and the rest of the system, namely the instruction scheduler, the operand storage component(s), and the power gating controller. For the baseline system, the capability to power gate the functional unit introduces four additional signals marked by dashed lines in Figure 5-8. The $FuncUnitX\text{Ready}/PowerGated$ signal informs the scheduler whether a power gateable unit is available for being issued an instruction during a cycle. The $WakeUpFuncUnitX$ signal is used by the scheduler to wake up a power gated functional unit when there is work available for it. The other two signals represent the interface between the power gating scheme and the functional unit.
5.5.1 Verification implications of adding power gating capability for functional units

The key to understanding why power gating functional units does not greatly increase microarchitectural design verification effort, with respect to a design without power gating, is that the scenario of a functional unit being unavailable due to power gating is very similar to the scenario in which the scheduler cannot issue an instruction to the functional unit due to a structural hazard. Assuming the initial processor design without power gateable units is correct, then its scheduler correctly identifies and handles the case when instructions with their operands ready cannot issue due to a structural hazard on the needed functional unit type. All that is necessary to maintain correctness, at the microarchitectural level, after the addition of power gating, is for a new structural hazard signal to be generated by OR-ing the old signal with the \textit{FuncUnitXPowerGated} signal.

The difference between unavailability due to a structural hazard and unavailability due to power gating is that we already know that a structural hazard will resolve. Thus, to make the verification equivalent, we must also verify that power gated units will eventually wake up, which involves verifying two properties:

- Property 1: We must verify that the scheduler correctly issues a wake-up signal to a unit when an instruction is ready for it and no other same-type units are available. A corollary is that the scheduler correctly identifies when a unit is unavailable to be issued instructions due to power gating.
• Property 2: The functional unit’s power gating control logic needs to be verified to ensure that the functional unit always wakes up from a power gated state within a fixed number of cycles after the scheduler asserts the wake-up signal.

Verifying these two properties would not add a significant burden to microarchitectural level design verification because they can both be verified in isolation. Verifying Property 1 involves the scheduler in isolation, by considering every possible value of its output \( \text{WakeUpFuncUnitX} \) and input \( \text{FuncUnitXReady/PowerGated} \) interface signals. Verifying Property 2 involves only the functional unit controller and would not translate into a system level verification effort.

5.5.2 Verification implications of adding the success monitor and token counting mechanisms

We now analyze the impact of our proposed Success Monitor Switch and Token Counting Guard on verifying the processor’s functional correctness, again at the microarchitectural level. The design we compare to is one that already enables power gating of processor execution units. We consider this initial design to be correct.

The mechanisms proposed in this work are illustrated in Figure 5-8 by the extra block on top of the Power Gating Algorithm (in bold). This level of Power Gating Monitor and Control interacts only with the Power Gating Algorithm block by reading the values of the efficiency counters and enabling or disabling the Power Gating Algorithm. Hence, our mechanisms do not change the initial interface between the functional unit and the rest of the system. The only change that occurs is that the Power
Gating Algorithm can be disabled in which case the \textit{PowerGateFuncUnitX} remains unasserted even when the unit is idle for longer than the idle\_detect. The initial design needs to respond correctly to both an asserted or unasserted value of the \textit{PowerGateFuncUnitX} signal, regardless of the length of its prior idle interval. Because of this, the addition of our mechanisms will not affect the functional correctness of the system compared to an initial design that allows for power gating of functional units.

### 5.6 Evaluation methodology

In this section we present our simulation methodology and the energy model used throughout our experiments.

#### 5.6.1 Simulation methodology

To evaluate the results of applying our proposed mechanisms for power gating, we use processor functional units’ utilization traces. Figure 5-9 depicts the generation of these traces and how we utilize them. To derive the unit-level utilization traces, we instrument a very detailed, cycle-accurate PowerPC simulation framework that was pre-configured to model an aggressive superscalar microprocessor core.

A high-level summary of the simulated microarchitecture is shown in Table 5-2. The model captures the details of the published core pipeline depth and superscalar
execution semantics of each core within the POWER6 microprocessor [49]. Our intent was to set up the baseline power-gating experimental analysis using the IdleCount algorithm proposed by Hu et al. [36], while upgrading from an older POWER4-like microarchitecture (as used by Hu et al.) to a more contemporary, POWER6-like model. We believe this new setting allows us to assess the leakage energy savings for the baseline power-gating algorithm as well as our new guarded, 2-level algorithms in a more realistic, modern setting.

Figure 5-9: Experimental framework.

---

3 The results and analysis presented in this work are not claimed to be an accurate representation of any real PowerPC product in the marketplace. In fact, as evident from published papers [49], power-gating features are not part of server-class processor core logic used in designs like POWER6.
Table 5-2: Target System Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional units</td>
<td>2 FXU, 2 FPU, 2 LSU</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>16K-entry BHT with 2 bits per entry to indicate direction of branch</td>
</tr>
<tr>
<td>Instruction buffer</td>
<td>64 entries for each thread</td>
</tr>
<tr>
<td>Decode width</td>
<td>8</td>
</tr>
<tr>
<td>L1D cache (within core)</td>
<td>64 KB, 8-way, line size 128 bytes</td>
</tr>
<tr>
<td>L1I cache (within core)</td>
<td>64 KB, 4-way, line size 128 bytes</td>
</tr>
<tr>
<td>L2 cache (on-chip)</td>
<td>4 MB, 8-way, line size 128 bytes</td>
</tr>
<tr>
<td>L3 cache (off-chip)</td>
<td>16 MB, 16-way, line size 128 bytes</td>
</tr>
</tbody>
</table>

The simulator is instrumented such that it reports the utilization of each processor unit on a cycle-by-cycle basis. Each row in Figure 5-9 represents the utilization of all units during one cycle (a record in the trace). A value of one signifies that the unit was used during that cycle while a zero means the unit was idle. Each column holds the utilization of one particular unit for successive cycles. We then implemented our power gating algorithms and proposed bounding mechanisms in an analysis tool written in the C language. This analysis tool implements an energy model for power gating that allows us to obtain estimates of leakage energy savings. A limitation of this trace oriented framework is that it allows us only an approximate analysis of the potential performance impact a power gating scheme could have. We assign a fixed cost in terms of cycles lost whenever a power gated unit is awoken after being power gated. In our experiments we assign a 3-cycle performance penalty in this situation. However, in practice this wake-up penalty can be largely avoided. In many cases it is possible to know, at the decode stage or from the issue queue, that a particular type of instruction is likely to execute within a
predefined number of cycles. If decode-time signals are properly utilized towards this end, it is possible to wake-up the units proactively when they are needed, avoiding the performance cost altogether. However, we consider both the case when such a decode mechanism is present and the case when it is not, by assigning a penalty of zero or three cycles at wake-up. The advantage of our analysis framework is that it allowed us to decouple the detailed, cycle-accurate performance model from the power-gating algorithm simulation, in such a manner that did not require us to modify the complex model semantics that were developed by a seasoned performance team. Only simple instrumentation code to monitor utilization traces for targeted units was required. We obtained traces of all SPEC2006 benchmarks (12 integer and 17 floating point) from the cycle-accurate PowerPC processor simulator previously described.

5.6.2 Energy Model

For estimating the amount of energy saved by our proposed schemes, we refer to the energy model proposed by Hu et al. [36]. From equation (11) in Hu et al. [36], the energy savings in the $i^{\text{th}}$ cycle after power gating and before the virtual $V_{\text{dd}}$ is fully discharged, $E_{\text{cyc saved}}$, can be described by the formula:

$$E_{\text{cyc saved}} = E_{\text{cyc}} * i * (\text{DIBL} / mV_t) * \Delta V_{\text{cyc}}$$

where $E_{\text{cyc}}$ is the leakage energy of the unit during one cycle, $i$ is the number of cycles after power gating, DIBL is the drain-induced barrier lowering factor, $m$ stands for subthreshold swing coefficient, $V_t$ is the thermal voltage, and $\Delta V_{\text{cyc}}$ represents the
voltage droop during the first cycle that the circuit is power gated. For current technology parameters, we obtain \( E_{\text{cyc saved}} = 0.045^* i^* E_{\text{cyc}} \).

The aggregate energy saved during the first \( N \) cycles after power gating and before full discharge, \( E_{\text{cyc saved}}^N \), is the sum over \( i \) of \( E_{\text{cyc saved}}^i \). After the virtual \( V_{\text{dd}} \) is fully discharged, the energy savings during one cycle, \( E_{\text{cyc saved}}^i \), equals the leakage energy of that component. Using the analytical model proposed by Hu et al. [36] for current technology parameters, we obtain a break-even point of 19 and an energy overhead of \( 8^* E_{\text{cyc}} \). This overhead represents the energy penalty paid each time power gating is applied. We make one small departure compared to the energy model we have been referring to. Hu et al. [36] observe that the point of full discharge is not identical with the break-even point for the precise values of the constants used. For simplicity, we approximate that the two are the same and that the average energy savings after the break-even point, \( E_{\text{cyc saved}}^i \), equals \( E_{\text{cyc}} \). The effect is that we use a value of \( E_{\text{cyc}} \) instead of \( E_{\text{cyc}}^i \) \( \times 0.9 \) for the 20th cycle after power gating and we use a value of \( E_{\text{cyc}} \) instead of \( E_{\text{cyc}}^i \) \( \times 0.945 \) for the 21st cycle. This small approximation permits us to use a single energy savings value for every cycle after the break-even point and we believe the difference to be in the noise margin of the model used by Hu et al. [36].

We have now obtained all values required to calculate the global energy savings of the power gating schemes in \( E_{\text{cyc}} \) units of measurement: the energy saved during cycle \( i \) after power gating and before the break-even point is \( E_{\text{cyc saved}} = 0.045^* i^* E_{\text{cyc}} \); the
energy savings during cycle $i$ after the break-even point is $E_{\text{cyc saved}} = E_{\text{cyc}}$; and the energy overhead is $E_{\text{overhead}} = 8* E_{\text{cyc}}$.

### 5.7 Experimental results

In this section, we first evaluate the ability of our mechanisms to bound the worst-case energy penalty of applying power gating (Section 5.7.1). For this purpose we show the impact of Success Monitor Switch and Token Counting Guard on the worst-case energy savings that were presented in Figure 5-5. We also analyze their impact on the average energy savings of power gating (Section 5.7.2) and on the average performance loss (Section 5.7.3). The power gating solutions we compare are the following:

- The baseline IdleCount solution proposed by Hu et al. [36] that we evaluate for 2 values of the IdleCount idle_detect (5 and 15, data presented for the 5 value).

- IdleCount augmented with the Success Monitor Switch (Success_IdleCount in the figures). The Success Monitor Switch is invoked every 50 cycles and evaluates whether the IdleCount solution is saving or losing energy. In case of loss, the IdleCount is disabled during the next 50-cycle monitoring interval.

- IdleCount augmented with both the Success Monitor Switch and the Token Counting Guard (Token_IdleCount). The token bag is updated every 50 cycles and the bound is set for a maximum of 2% leakage energy loss over a guarantee interval of 100 monitoring intervals (a total of 5000 cycles).
• The last comparison point implements both the Success Monitor Switch and the Token Counting Guard (TokenSuccess_IdleCount). This compound solution disables the IdleCount power gating in two scenarios: when the Success Monitor estimates that the IdleCount wasted energy during the last 50 cycles, and when the token bag does not have enough tokens left to tolerate the worst-case energy loss scenario during the next 50 cycles.

In the experiments below we consider the break-even point to be 19 and we vary the number of wake-up penalty cycles from one to three. We present the performance impact of the power gating schemes relative to the penalty incurred by the baseline. In reporting energy savings, we explicitly consider the leakage energy consumed during cycles added due to power gating.

**5.7.1 Worst-case energy penalty**

Figure 5-10 compares the energy savings for all schemes described above for the benchmarks that exhibited an energy penalty under the baseline IdleCount scheme (the IdleCount data was presented in Figure 5-5). There are two bars presented for each of the compared schemes. The solid bar represents the energy savings for a system that masks the wake-up penalty while the striped bar represents the energy savings of a system with a 3-cycle wake-up penalty. We notice that our Success_IdleCount proposed solution eliminates the energy penalties incurred by the baseline system and transforms them into energy savings for all benchmarks.
Figure 5-10: Comparative energy savings for the benchmarks wasting energy under the IdleCount scheme.

TokenSucc_IdleCount maintains a marginal energy penalty of below 1% only for *hmmer*. Token_IdleCount exhibits a marginal energy penalty of below 1% for the *leslie3D*, *gobmk*, and *hmmer* benchmarks. Note that the bound that was set for both Token_IdleCount and TokenSucc_IdleCount schemes was a maximum loss of 2%, which means they respect the bound and function correctly.

### 5.7.2 Average-case energy savings

Figure 5-11 and Figure 5-12 compare the average energy savings for an idle_detect threshold of 5 and a break-even point of 19 (for SPECint and SPECfp). The graphs on the left show data when we consider the performance penalty to be zero while the right figures show energy savings for a 3-cycle penalty on unit wake-up. We notice that in all four figures the Token_IdleCount scheme saves slightly more energy than the baseline. The same is true for the Success_IdleCount and
TokenSuccess_IdleCount except for the SPECfp benchmarks for the 0-cycle performance penalty case.

Figure 5-11: Average energy savings (SPECint)

Figure 5-12: Average energy savings (SPECfp)

A trend can be seen where our proposed schemes perform increasingly better compared to the baseline when there is a performance penalty paid. The explanation is that our schemes incur a smaller performance penalty due to the fact that power gating is disabled when it is not useful. Due to the smaller performance penalty of our schemes
the supplementary execution cycles due to power gating have a lower effect on our mechanisms. For a larger idle_detect of 15, our schemes are still marginally advantageous in terms of average energy savings. Averaged across both values of the idle_detect, the Token_IdleCount saves 3.5% more energy; the TokenSuccess_IdleCount saves 2% more energy while the Success_IdleCount saves 1.3% less energy than the IdleCount baseline. These improvements in energy saving are only marginal. We present them to show that our proposed schemes are capable of successfully bounding the worst-case energy penalty without degrading the average case energy savings.

5.7.3 Performance

Figure 5-13 and Figure 5-14 show the worst-case potential performance impact of applying power gating. We call it potential impact because much of it is likely to be masked in a real processor due to other causes of stalls. All our proposed schemes decrease the baseline performance impact to a large extent. This decrease occurs because in many cases our schemes disable power gating when it was incurring an energy penalty; this disabling of power gating when it is not beneficial leads to a decrease in performance penalty as well. On average, the Token_IdleCount decreases the baseline performance impact by 51%, the Success_IdleCount by 71% and the TokenSuccess_IdleCount by 63%. In addition to the bound on the worst-case energy behavior we thus also obtain a significant decrease in average performance
impact.

Figure 5-13: Worst-case Potential Performance Impact (SPECint).

Figure 5-14: Worst-case Potential Performance Impact (SPECfp).

5.8 Related work

There has been prior work in power gating, energy management, and speculation control.
5.8.1 Power gating mechanisms and algorithms

Intel’s recent Nehalem processor incorporates power gating techniques at the core granularity (instead of the finer functional unit granularity we consider) [77]. Because Nehalem’s power gating control algorithms have not been published, we cannot estimate whether adding our guard mechanisms would be beneficial.

Rele et al. [71] propose a combination of compiler and hardware support for power gating processor functional units. The compiler attempts to extract maximum energy savings opportunities from code by providing power gating hints to the hardware about regions where functional units are expected to be idle. The hardware follows these hints unless the turn-on hint occurs while the turn-off instruction is still executing. Without a guard mechanism, it is still possible for this power gating solution to waste a significant amount of energy. This can happen if the turn off/on hints are a distance that is slightly longer than the execution time of the turn-off hint instruction. Youssef et al. [87] propose an incremental improvement to the prior single-level IdleCount algorithm, in which the idle_detect threshold is adaptively set. Drophso et al. [29] propose power gating functional units in gradual steps equal in number to the break-even point. Their scheme can still waste energy for repetitive patterns that contain small idle intervals. Mohamood et al. [61]propose a queue-controller scheme that bounds the worst case di/dt overshoots that can be caused by simultaneous switching due to applying power gating techniques. We also target worst case behavior due to
power gating but our mechanisms are necessarily different to permit us to estimate at runtime the win or loss of power gating policies. Jiang et al. [43] evaluate the benefits and costs of implementing power gating from a circuit-level perspective.

Our focus is on microarchitectural level mechanisms for ensuring that the use of power gating does not translate into a significant energy penalty for various runtime applications.

### 5.8.2 Energy management

Zeng et al. [88] introduce *currentcy* as a unifying abstraction used by the operating system for the management of devices that consume energy. Applications are allocated a certain currentcy that they can use during an epoch to satisfy their energy requirements. At a high level, the currentcy abstraction is similar to our token mechanism; however, we use tokens to quantify an undesired behavior of a power management policy (incurring an energy penalty). Also, we use the token abstraction to provide guarantees on the worst-case behavior of a power gating algorithm.

### 5.8.3 Speculation control

Speculation-control methods have been used in conjunction with branch predictors in order to save energy [41]. However, the power gating problem domain dealt with in this work is quite different. If we do not have a guard mechanism, such as the Token Counting Guard we propose, the original goal of energy reduction itself may be reversed into a net energy increase situation. In contrast, the primary goal in branch
prediction is performance increase, and the speculation control mechanism is used to save wasted energy, at a small performance cost.

5.8.4 Other mechanisms for reducing leakage

Input vector control [86] applies an input pattern to a combinatorial circuit so as to decrease the leakage power while maintaining the same functional behavior. Adaptive body bias [80] is a post-silicon technique that attempts to minimize the process variation effects on leakage power. These circuit-level schemes are complementary to power gating.

Our work differs from all of the above through our focus of ensuring that a power gating scheme does not become a vulnerability of the system by being transformed, at runtime, into a significant source of energy loss. The Token Counting Guard mechanism we propose increases the overall quality of the power gating design because it ensures the control hardware is “safe” in the sense that power overruns are prevented.

5.9 Conclusions

Power gating is a promising technique for reducing leakage energy consumption, yet it has potential drawbacks that, if not overcome, could preclude its use. In this chapter, we have presented two mechanisms for overcoming these problems. The key result is that we can bound the worst-case energy penalty incurred by a power gating scheme, with minimal impact on power gating’s average energy savings.
Furthermore, we address the important issue of verification by showing that our mechanisms add little complexity to the processor verification process. We believe that our mechanisms, plus the verification analysis, reduce the barrier to adopting power gating and make it an attractive solution for reducing leakage energy. An interesting avenue of future work is applying our mechanisms to speculative techniques other than power gating. Our mechanisms apply generally, and they can be used to reduce the risk of other speculative techniques that have high upsides but large potential losses in the case of excessive mispredictions.
6. Summary and Conclusions

Technological advances and engineering ingenuity have enabled computers to become present and sometimes indispensable in many application domains (finances, medical areas, transportation, etc.). The fact that our society relies more and more on computers for deriving, processing, and preserving its critical information has led to indisputable benefits, but also to high requirements for computing systems to function correctly, always. Although computers can err for a variety of reasons (fabrication defects, soft errors, hard errors, etc.) the impact of design errors can be truly catastrophic because all produced units are affected.

The ever increasing complexity of current processors, coupled with small time to market, make design verification an extremely difficult task which consumes the majority of the resources allocated to a new processor design. In the current design flow, verifiability is usually not explicitly considered at an early design stage - when design decisions are most influential - because at that point only more traditional goals are prioritized such as performance, power, and area. The resulting design might thus prove very difficult to verify in the end, precisely because its verifiability was not a main design target in the beginning.

In this thesis we propose to view verifiability as a critical design constraint that is considered, together with other important metrics (like performance, power), from the initial stages of design. Our high-level goal for this strategy is to lead to processor
designs that are easier to verify, thus decreasing the difficulty of and resources consumed by the design verification step. Arguing for this change in perspective, namely the rearrangement of processor design goals to be pursued at an early design stage, constitutes the first contribution of this thesis.

The second contribution of our thesis is that of quantitatively evaluating the impact of various processor design choices on design verifiability. More specifically we select a subset of processor design areas (parts of the processor core, organization of the memory hierarchy, power management techniques) and analyze how the desirability of one design option over another is transformed by considering verifiability as a critical design constraint. We conclude that small differences in microarchitecture may lead to significant differences in verification effort. For example, increasing the pipeline depth by one stage significantly increases verification effort, although the exact impact also depends on whether the extra stage occurs in the beginning or end of the pipeline. Design options that decrease the amount of symmetry in the processor (like address fragmentation through the memory hierarchy) negatively impact the verification process. These results confirm the hypothesis that tiled architectures hold the promise not only of reduced power envelopes but also of significant benefits for verification.
The third contribution of this thesis is that we investigate trade-offs between a design’s verifiability and the more traditional quality metrics of performance and power. We investigate power management through DVFS in the context of multi-core processors and show that both global controllers and fine-grained frequency tuning greatly increase verification effort while adding little benefits for performance or power savings goals.

Our fourth contribution is to re-design several components for increasing their verifiability. We show how specific choices for the block size and associativity across cache levels in the memory hierarchy can improve symmetry of the address fields and improve verifiability. For power management through speculative power gating, the actual power savings of the DPM depends on the runtime behavior of the applications. We propose to first define an explicit specification for what constitutes correctness for these DPM schemes in terms of the maximum power loss acceptable due to the DPM. We then propose a mechanism that bounds the worst case power penalty that can be incurred at runtime by power gating. Having this bounding mechanism in place guarantees that the power gating scheme meets its correctness specification. For CMP power management with DVFS, we propose designs that group a small number of cores under the management of a single controller as the solution that best balances verification effort, power and performance goals.
As our technological capabilities advance, computing systems are likely to pervade even more application domains and have even more complex designs. Given the importance and difficulty of design verification, we believe that considering verifiability as a design priority to be addressed in the early stages of design has the potential to lead to designs that are indeed easier to verify and ultimately more robust.
References


35. Hosabettu, R., Gopalakrishnan, G. and Srivas, M. Verifying Advanced Microarchitectures that Support Speculation and Exceptions. CAV. 521-537.


37. Hum, R. How to Boost Verification Productivity. EE Times.


76. Shyam, S. and Bertacco, V., Distance-Guided Hybrid Verification with GUIDO. in *Design Automation and Test in Europe (DATE)*, (Munich, Germany, 2006).


81. Wagner, I. and Bertacco, V. Engineering Trust with Semantic Guardians 
Proceedings of the Conference on Design, Automation and Test in Europe, EDA 

82. Wagner, I., Bertacco, V. and Austin, T. Shielding Against Design Flaws with 
Field Repairable Control Logic Proceedings of the 43rd Annual Conference on Design 

83. Wagner, I., Bertacco, V. and Austin, T., StressTest: An Automatic Approach to 
Test Generation Via Activity Monitors. in Design Automation Conference (DAC), 
(Anaheim, CA, 2005).

84. Waiingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., 
Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. and Agarwal, A. Baring 
it All to Software: Raw Machines. IEEE Computer, 30 (9). 86-93.

85. Wang, N.J., Quek, J., Rafacz, T.M. and patel, S.J. Characterizing the Effects of 
Transient Faults on a High-Performance Processor Pipeline Proceedings of the 2004 
International Conference on Dependable Systems and Networks, IEEE Computer 

86. Ye, Y., Borkar, S. and De, V., A New Technique for Standby Leakage Reduction 
in High Performance Circuits. in Proceedings of the Symposium on VLSI Circuits, 

Tolerant Microprocessor Functional Units Proceedings of the 39th Annual 
IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 
2006.

as a First Class Operating System Resource. in Proceedings of the Tenth 
International Conference on Architectural Support for Programming Languages and 
Biography

Anita Lungu was born on February 21st, 1980 in Craiova, Romania. She received her B.Sc. in Computer Science from University of Craiova, Romania in 2003 and her M.S. in Computer Science from Duke University in 2007. She was a recipient of a Diploma of Excellency for outstanding academic results in 2003, an Outstanding M.S. Thesis Award in 2007, and an IBM PhD Scholarship in 2008. Anita Lungu’s research interests include the intersection between computer architecture and formal verification methods, in particular the area of verification-aware processor design. Her Ph.D. thesis explores the impact of various processor design choices on verifiability, analyzes trade-offs between verification effort and other traditional metrics like power and performance, and proposes design changes to increase a design’s verifiability.