University of California
Santa Barbara

Understanding and Estimating Uncertainties and Risks in Architecture Design: from Analytical Analysis to Detailed Simulation

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Weilong Cui

Committee in charge:

Professor Timothy Sherwood, Chair
Professor Frederic T. Chong, University of Chicago
Professor Rich Wolski
Professor Yuan Xie

September 2019

The Dissertation of Weilong Cui is approved.

Professor Frederic T. Chong, University of Chicago

Professor Rich Wolski

Professor Yuan Xie

Professor Timothy Sherwood, Committee Chair

August 2019

Understanding and Estimating Uncertainties and Risks in Architecture Design: from Analytical Analysis to Detailed Simulation

by

Weilong Cui

To Mom, Dad and my loving wife, Yan Tang.

Acknowledgements

I'll try to make this in one page.

I am eternally grateful to Pro¢fessor Tim Sherwood. He is a one-of-a-kind advisor to me. His broad vision, knowledge and unconditional support encourage me to take risk (literally) as I explore directions for research. His graduate-student-like coding passion and totally-not-graduate-student-like writing standard influence me so much over the years and will be kept as a role-model in the many years to come. His complete lack of micromanagement skills makes my phd journey so surprisingly enjoyable and fruitful that I only stressed out like once or twice (by myself, not because of him) in 5 years. He is also much more than an advisor to me. His incredible friendliness and infectious humor make a second home out of a distant place and a foreign culture for me. Especially, his thoughtful and caring personality keeps my wife and I from being geologically separated.

I am also grateful to my entire doctoral committee. I am so honored to have you be there throughout the journey. I am thankful to Professor Fred Chong for picking me out of the pile of phd applicants 5 years ago and the great support when I first came to this country. I am thankful to Professor Rich Wolski for his unparalleled enthusiasm and insights about our field which inspire me many many times and for the best operating system lectures ever. I am thankful to Professor Yuan Xie for his kind encouragement, generous support and most valuable advice every time I am challenged with a major career choice.

In the end, I would also like to thank all my colleagues and collaborators, Prof. Zheng Zhang, Dr. Georgios Michelogiannakis, Dr. Dilip Vasudevan, Zichang He, Dr. Chunfeng Cui, Dr. Lunkai Zhang, and my dearest lab-mates and friends, George Tzimpragos, Deeksha Dangwal, Joseph mcMahan, Alvin Oliver Glova and Dr. Nestan Tsiskaridze, for your hard work, help and inspiration behind this thesis.

Curriculum Vitæ

Weilong Cui

EDUCATION

2019	Ph.D. in Computer Science (Expected), University of California, Santa Barbara
2014	M.S. in Computer Science, Peking University
2011	B.S. in Computer Science, Peking University
2011	B.E. in Economics (double major), Peking University

RESEARCH EXPERIENCE

- **ArchLab** **UC Santa Barbara**
  PhD student, advisor: Tim Sherwood Oct.2014 - Present
  Thesis: "Understanding and Estimating Uncertainties and Risks in Architecture Design: from Analytical Analysis to Detailed Simulation".

  Cross-domain architecture analysis with risk assessment.
  Programming language support for analytic modeling.
  Simulation techniques to support scalable UQ in architecture.
  Information leakage quantification of SRAM cells.

- **Computer Architecure Group** **Lawrence Berkeley National Labs**
  Visiting researcher, host: George Michelogiannakis July.2018 - Present

  Quantifying uncertainty in detailed simulations.
  Sensitivity analysis of CMP design.

- **CMRR** **UC San Diego**
  Visiting student, advisor: Vitaliy Lomakin Sept.2013 - Nov.2013

  Parallelization of micro-magnetic simulation and FFT kernels on GPUs.

- **Parallel Algorithm Research Group** **Peking University**
  MS student, advisor: Yifeng Chen Sept.2011 - Jul.2014

  Fine-grained parallel memory architecture.
  Parallelization of graph algorithms (MST, kd-tree, etc).

- **National Research Center for Software Engineering** **Peking University**
  Research intern, mentor: Yu Huang Mar.2010 - Dec.2011

  Design/implementation of internal workflow management system.

AWARDS

- 2019 Outstanding Publication Award. **UC Santa Barbara**

ACADEMIC ACTIVITIES

- Reviewer for ACM Transactions on Architecture and Code Optimization (TACO).
- Reviewer for Elsevier Journal on Microelectronics.
- Reviewer for Springer Journal on Hardware and System Security.

TEACHING/MENTORING EXPERIENCE

- **Mentoring Undergrad Research** **UC Santa Barbara**
  Mentor to Yu Tao Fall 2018

  Task design, 1:1 discussion and hands-on programming sessions.
- **CS 254: Computer Architecture** **UC Santa Barbara**
  Teaching assistant Spring 2016

  TA, assignments
- **CS 154: Computer Architecture** **UC Santa Barbara**
  Teaching assistant Spring 2015

  Lead TA, programming assignments.
- **CS 64: Computer Organization** **UC Santa Barbara**
  Teaching assistant Winter 2014

  Lead TA, lab sessions.
- **CS 64: Computer Organization** **UC Santa Barbara**
  Teaching assistant Fall 2014

  TA, Lab sessions.

INDUSTRY EXPERIENCE

- **Performance and Simulation Team** **Google, Mountain View, CA**
  SWE intern, mentor: Slava Malyugin June.2016 - Sept.2016

  Post-assembly optimization of Android application for ARM in-order CPUs.
- **Platform Team** **Google, Madison, WI**
  SWE intern, mentors: Dan Gibson and Christopher Alfeld June.2015 - Sept.2015

  Benchmarking High-performance networking protocol in datacenters.

PUBLICATIONS

- Zichang He, Weilong Cui, Chunfeng Cui, Timothy Sherwood, and Zheng Zhang. Efficient Uncertainty Modeling for System Design via Mixed Integer Programming Proceedings the International Conference On Computer Aided Design. (ICCAD) November 2019. Westminster, CO.
- Deeksha Dangwal, Weilong Cui, Joseph McMahan and Timothy Sherwood. Safer Program Behavior Sharing Through Trace Wringing to appear in Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2019, Providence, RI.
- Weilong Cui, Yongshan Ding, Deeksha Dangwal, Adam Holmes, Joseph McMahan, Ali JavadiAbhari, George Tzimpragos, Frederic T. Chong and Timothy Sherwood. "Charm: A Language for Closed-form High-level Architecture Modeling" in Proceedings of the International Symposium of Computer Architecture (ISCA) June 2018. Los Angeles, CA.
- Weilong Cui and Timothy Sherwood. "Architectural Risk" in IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, January-February 2018.
- Weilong Cui and Timothy Sherwood. "Estimating and Understanding Architectural Risk" in Proceedings of the International Symposium on Microarchitecture (Micro) October 2017 Boston, MA.
- Joseph McMahan, Weilong Cui, Liang Xia, Jeff Heckey, Frederic T. Chong and Timothy Sherwood. "Challenging On-Chip SRAM Security with Boot-State Statistics" in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), (short paper) May 2017 Washington D.C.
- Sidi Fu, Weilong Cui, Matthew Hu, Ruinan Chang, Michael J. Donahue, Vitaliy Lomakin. "Finite-Difference Micromagnetic Solvers With the Object-Oriented Micromagnetic Framework on Graphics Processing Units" in IEEE Transactions on Magnetics, vol. 52, no. 4, pp. 1-9, April 2016.

TALKS

- "Charm: A Language for Closed-form High-level Architectural Modeling", ISCA, 2018, Los Angeles, CA
- "Estimating and Understanding Architectural Risk with Analytic Models", MICRO, 2017, Boston, MA.
- "Parallel Memory Access with Fine-grained Transaction Management", Google, 2015, Madison, WI.

Abstract

Understanding and Estimating Uncertainties and Risks in Architecture Design: from Analytical Analysis to Detailed Simulation

by

Weilong Cui

Computer architecture is always changing. Now, more than ever, we see deeper vertical integration with domain-specific software, faster emergence of paradigm shifting computing devices and memory technologies, and unprecedented security and privacy vulnerabilities. These changes all present opportunities for innovations in architecture design, and along with them, uncertainties we have to deal with carefully. An uncertainty-aware approach is essential when we design computer architectures and systems for the future, as variations in technology alone has been demonstrated to have the potential of eliminating the performance gain of an entire CMOS generation. To navigate the new dark waters exposed by changes in fabrication, design constraints, and programming models, a clear understanding and rigorous quantification of uncertainties in architecture designs, as well as the risks that come along, is a critical first step.

To achieve such a goal in a tremendous space of designs, from the smallest embedded system to the largest warehouse-scale computing infrastructure, from the most well-characterized CMOS technology node to novel devices at the edge of our understanding, we need new systematic supports across the design stack from high level analytical analysis when evaluating design decisions in the early stage to cycle-accurate detailed simulations when projecting actual system performance.

This thesis establishes a route to build a new uncertainty and risk aware architecture design process. We first demonstrate how even a very high level definition and understanding of such

concepts of uncertainties and risks can expose a new trade-off space between average-case performance and the amount of uncertainty/risk a design is exposed to. The framework is then generalized and we design a new modeling language that systematically support such analysis through a combination of symbolic execution, graph transformation and compiler optimizations, followed by demonstration of the applicability and benefits of such a modeling language as a foundation to build high quality analysis with closed-form models. We then take the exploration down to the most common practice of architecture studies: cycle-accurate simulations. There a new way of quantifying uncertainty and risk while keeping the computational taxing at bay is proposed through the adaptation of generalized polynomial-chaos theories to build accurate surrogate models.

Finally, we peak into the future and envision what needs to be researched before quantifying uncertainties and risks become an essential and well-supported practice of architecture design in this new golden era.

Contents

Chapter 1

Introduction

The deep vertical integration of domain-specific software and hardware (e.g., [1]), the emerging paradigm-shifting computing devices and memory technologies (e.g., [2,3]), as well as the recently revealed security and privacy vulnerabilities (e.g., [4,5]) all present opportunities for innovative architecture designs. However, this dramatic shift in the computing landscape also means that it is, more than ever, wrought with uncertainties.

Like in many other disciplines of science and engineering, uncertainty, as well as the risk it brings along, poses a serious threat to suffocate an entire new generation of technologies if not dealt with properly [6]. However, unlike other mature disciplines, these aspects of a system design, i.e., how uncertainty propagates and impacts system-level performance, often sneak under the radar of computer architects. Architects tend to focus on quantifying and optimizing performance metrics such as IPC, throughput, and power efficiency but generally fail to consider how *exposed* to uncertainty and risk a class of designs might be.

A clear understanding and rigorous quantification of uncertainties and risks in architecture design is critical to navigate the new dark waters exposed by changes in fabrication, design constraints, and programming models. Instead of leaving the problem to later design cycle such as logic and physical design, computer architects with the knowledge of much higher-level

system specs and constraints compared to circuit designers and much lower-level hardware understandings compared to application developers, have the unique opportunity to take uncertainties into consideration much earlier in the design cycle to understand their impacts and deal with the risks that come along proactively and effectively.

In order to quantitatively understand and deal with uncertainty and the risk it creates, both new concepts and new tools are required to answer challenging new questions including: how can we discern the uncertainties that will impact system behavior or performance from all that is unknown to us in design time; how can we define a measurable metric that reflects the potential loss from uncertainties; what is an uncertainty-aware analysis approach that is applicable to computer architecture design flow; and eventually what new understandings and system design guidance can we get from a quantitative uncertainty analysis?

1.1 Uncertainties and Risks in Architecture Design - Where We Are

Computer architecture has always been governed by a combination of physical laws, human creativity, and economic realities. The decision to invest the engineering hours, the design and test infrastructure, and the initial fabrication costs into a new design is never taken lightly. However, the lack of a clear forecast for both new technologies and new application domains means that this investment involves *significant* uncertainties. An example is the process variation in the CMOS manufacturing process. From the addition or deletion of a few impurity atoms known as random dopent fluctuation to the temperature and humidity differences in the ambient environment, transistors are not born equal; they come with slightly different geometries and electrical properties, and when billions of such small differences add up, a system-scale behavior change can emerge [7]. These variations or uncertainties are not limited to the

hardware. Randomness across the software stack from OS scheduling to diversified and unseen workloads/inputs also plays an important role in determining the actual performance of a system in the field.

Figure 1.1 shows a high-level view of a typical ASIC design flow. It usually starts from an application-level analysis of the target workloads. The analysis results inform later system design of what type of architecture to consider. The architecture design process includes high-level decision-making (e.g., deciding what type of system to design) defining high-level system constraints, evaluating IPs for their applicability, and estimating performance through system modeling. Once a design candidate is chosen, it goes into the hardware design and verification process.

Most existing work characterizes and evaluates uncertainties after a design has been taped-out. Some uncertainty such as process variation (PV) is well-studied for mature technologies like CMOS [8]. Whether it's die-to-die (D2D) or within-die (WID), most work focus on characterizing a certain uncertainty source, e.g., process/supply voltage/temperature (PVT) variation for transistors [9], frequency variation for chips [10]. The variation in workloads for popular benchmark suits from different aspects are also well-studied [11–13]. Even the randomness in scheduling for multi-threaded workloads is examined [14, 15]. However, seldom do computer architects consider a quantified uncertainty analysis essential in early design time nor do we have a clear understanding of, nor an easy way to characterize, how these uncertainties would interact at the architecture level and impact system performance. Such a quantitative approach to study how exposed to risks our designs are, especially early in the design cycle when this information is most needed to guide high-level design decisions (e.g., what type of cores to integrate on a CMP) is mostly missing, only appear sometimes in an ad-hoc manner [16].

As architecture design orchestrates hardware and software, it is the place where difference types of uncertainties interact. Starting in Chapter 2, we identify the most important uncertainty sources and propose a taxonomy to help scoping such analysis on uncertainties in an

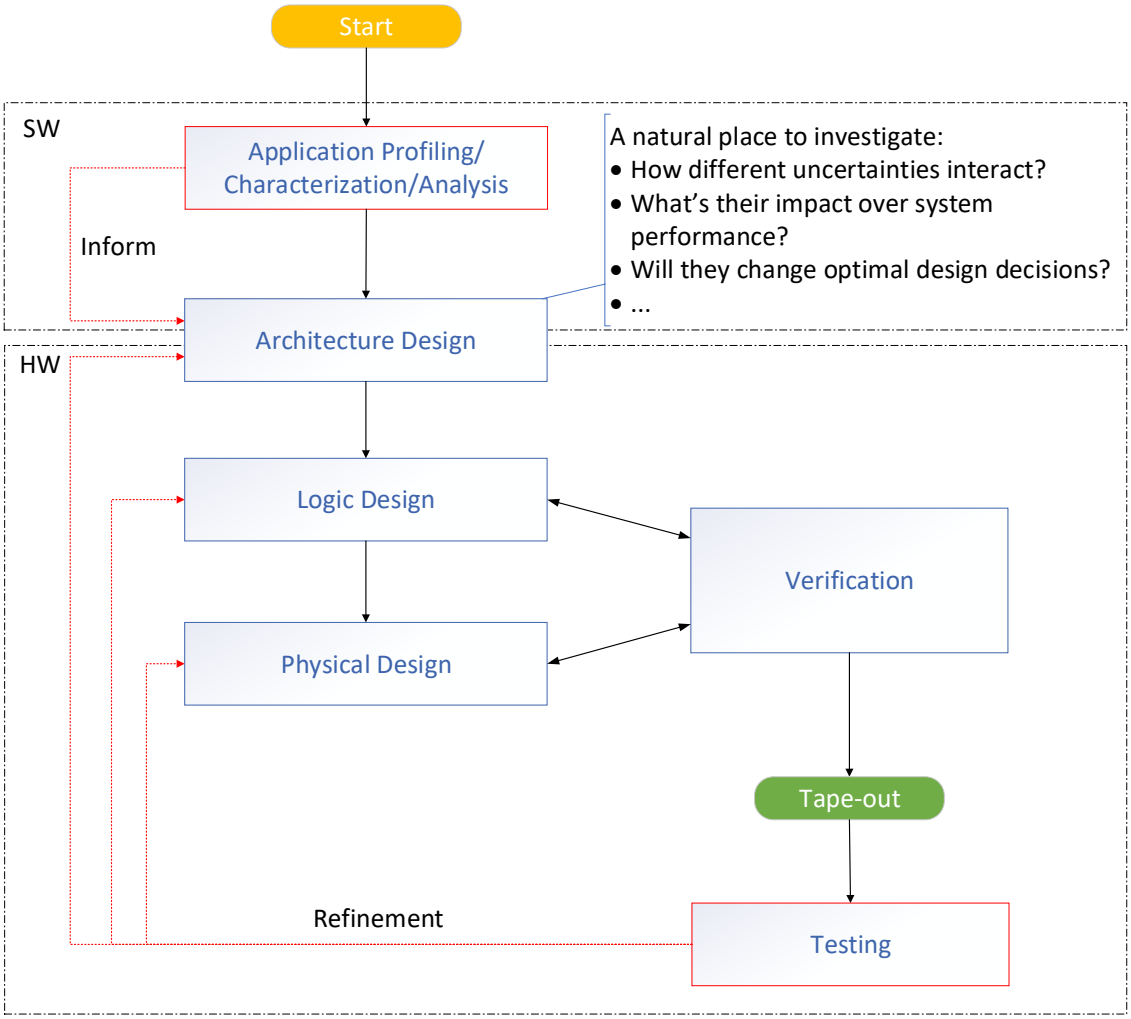


Figure 1.1: A high-level view of an uncertainty-aware chip design flow. Most existing works only investigate uncertainty after tape-out and reflect on the impact of uncertainty on designs very late in the design cycle. We argue that uncertainty quantification should be carried out in early architecture design stage.

architecture design space. In order to evaluate and analyze the uncertainty-induced risk, we need to identify what risk means in the context of computer system design and how it is measured such that the trade-offs between performance and risk can be explored quantitatively. As we explore in Chapter 2, even with fairly simple assumptions and conservative constraints, these new trade-offs can lead to surprising findings.

1.2 Thesis Statement

In this thesis, we make the following statement through experimentation and case studies with real-life multi-core processor architectures:

In a design space of ever-increasing complexity and uncertainty, it is possible to accurately quantify and analyze the propagated uncertainties through both high-level analytical models and detailed simulations to gain new understandings of the scaling trends and trade-offs between uncertainty, risk, and other aspects of the system. Such analysis can also be supported in a systematic and scalable way through adaptation of programming language design principles and applied surrogate modeling methods, respectively.

To support the above statement, we start with exploring a well-studied analytical model from the new perspectives of uncertainty and risk to discover interesting new trade-offs. We then propose a custom-designed modeling language to systematically support such studies before we move on to utilize advanced mathematical methods and the most well-trusted architecture simulators to build an analysis framework to estimate and manage uncertainties and risks with both high efficiency and good accuracy. The end goal of this thesis is to find new ways to guide future designs towards being more robust to the impacts of uncertainty than performance-only-optimal designs while still maintaining very strong performance in the common case.

1.3 Uncertainty Quantification of An Architecture Design - Methods and Challenges

In order to quantitatively study uncertainties and risk, we need some type of a system model to efficiently explore the vast system design space with both properties from hardware devices and characteristics from application workloads.

In this thesis, we argue that the system modeling and quantitative uncertainty and risk analysis should be carried out in early design cycles. As we will demonstrate in this work, the information we acquire from such analysis may justify or nullify important design assumptions and decisions (e.g., what type of memory technology to use) and hence, if deployed late in the design cycle, may result in a waste of engineering resources on designs that won't meet the end goal of the system, or too "risky" (we will discuss what this means in Chapter 2) after all.

Naturally, a good choice for architecture modeling is a cycle-accurate simulator. In fact, software simulators/emulators (e.g., gem5 [17]) are the most commonly used tools for computer architects to explore and evaluate their designs early in the design cycle. By simulating what hardware does functionally with accurate timing constraints, simulators faithfully capture how real systems behave and achieve good accuracy and offer good extensibility to cover most commonly seen types of technologies and architecture.

Although careful application of detailed simulation can accurately estimate the potential of a specific architecture design, it is far from being efficient enough to cover the vast design space when uncertainty is taken into consideration without innovative ways to accelerate such a process.

Another type of modeling commonly used when exploring higher level questions are analytic models, especially closed-form ones (e.g., the well-known Hill-Marty model [18]). Through simplification and approximation, these models are orders of magnitude more efficient to evaluate compared to simulators at the cost of accuracy and the level of details being captured by

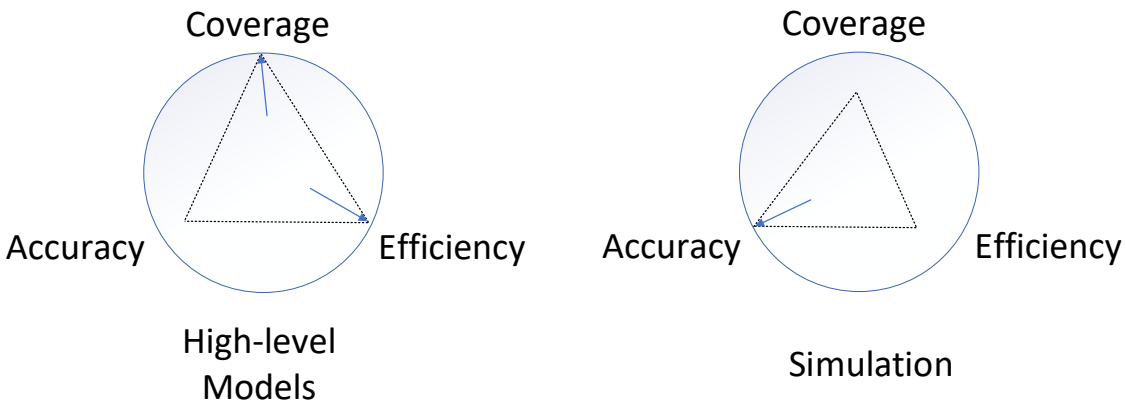


Figure 1.2: Comparison of analytical models and simulators at a high level.

the model. Figure 1.2 depicts the key differences between simulation and high-level models. Although not as accurate, closed-form models are still extremely useful to evaluate early design choices in relative terms and to explore fundamental trade-offs in the design space. For example, "given some target cooling budget, how much more performance can I get out of an ASIC versus an FGPA for this application given my ASIC will be 2 tech nodes behind the FPGA?"

The explosion of domain-targeted computing solutions means that more and more people are being asked to answer these questions and with some understanding of the confidence in those answers. While any Ph.D. in Computer Architecture should be able to answer such questions, when you break them down, they require a combination of a surprisingly complex set of assumptions. How do tech node and performance relate? What is the relationship between energy use and performance? ASIC and FPGA performance? Dynamic and leakage power? Temperature and leakage? Any result computed from these relationships will rely on the specific relationships chosen, on those relationships being accurate in the range of evaluation, on a sufficient number of assumptions being made to produce an answer (either implicitly or explicitly), and finally on that the end result be executable to the degree necessary to explore a

set of options (such as for a varying parameter e.g., total cooling budget).

Both closed-form analytical models and cycle-accurate simulators are essential to enable informed design decisions under the influence of uncertainties and risk. In this thesis, we identify the pain points of drawbacks of both methods and explore new ways of enabling and supporting efficient and scalable analysis frameworks for both types of models in Chapter 3 and Chapter 4 respectively.

1.4 Thesis Contributions

We make the following contributions in this thesis:

- In Chapter 2, we propose a new taxonomy to categorize different uncertainty sources in system design and, for the first time, formally define a general risk metric that captures the impacts of uncertainties in terms of system performance. With the introduction of architectural risk, we enable further analysis of uncertainties and their impacts on system designs be carried out in a quantifiable and easily understandable way.

- In Chapter 2, we, for the first time, demonstrate that a high-level analysis with closed-form models can reveal the new trade-off space between risk and performance even under conservative and simplified assumptions. We identify several interesting (some counter-intuitive) design guidelines for the core selection problem of a chip-multiprocessor design under the influence of uncertainties and risks.

- In Chapter 3, we, for the first time, design a declarative modelling language that is tailored to support high-level architecture performance reasoning through adaptation of programming language and compiler techniques. Charm, our proposed language, provides a new systematic way to express closed-form architecture models and high-level analysis in a clear, flexible, and optimized manner. More importantly, it enables collab-

oration between application developers, computer architects and hardware designers to proactively reason about properties of their work by building collectively a set of accurate and useful models via a common abstraction layer.

- In Chapter 4, we, for the first time, tackle the problem of quantifying uncertainty with detailed simulations by appling the theories of generalized polynomial chaos (gPC) to efficiently build surrogate models such that the computational taxing is kept at bay when the design space explodes with uncertainty. This innovative method enables large-scale, accurate and trusted performance analysis under uncertainty on the most widely used infrastructure for computer architects.

1.5 Thesis Outline

Chapter 1 discusses the idea of uncertainties and risk in the settings of computer architecture research and describes the status quo of both analytical architecture modeling and detailed simulations. Chapter 2 motivates the work and demonstrates the new knowledge and design guidance from a quantification of uncertainties and risks for the core selection problem in a multi-core processor at a high level. Chapter 3 generalizes the idea and framework in Chapter 2 and proposes a modeling language design that systematically support high level analytical analysis with uncertainties and risks, followed by Chapter 4 which takes the question of estimating and understanding uncertainties and risks in the most widely adopted settings of architecture study with detailed simulators and proposes a new way of accurately quantifying uncertainties and risks while incurring minimum computational cost.

1.6 Permissions and Attributions

1. The content of Chapter 2 is the result of a collaboration with Timothy Sherwood, and has previously appeared in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture [19]. It is reproduced here with the permission of ACM [1] and IEEE [2].

2. The content of Chapter 3 is the result of a collaboration with Yongshan Ding, Deeksha Dangwal, Adam Holmes, Joseph McMahan, Ali Javadi-Abhari, and Georgios Tzimpragos, Frederic T. Chong and Timothy Sherwood, and has previously appeared in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) [20]. It is reproduced here with the permission of ACM and IEEE.

3. The content of Chapter 4 is the result of a collaboration with Zichang He, Alvin Oliver Glova, Georgios Tzimpragos, George Michelogiannakis, Dilip P. Vasudevan, Zheng Zhang and Timothy Sherwood. Part of the content has previously appeared in 2019 IEEE/ACM 38th International Conference On Computer Aided Design (ICCAD) [21]. It is reproduced here with permission from IEEE/ACM and all co-authors.

[1] https://authors.acm.org/main.html
[2] https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/permissions_faq.pdf

Chapter 2

Estimating and Understanding Uncertainty and Architectural Risk at A High Level

In this chapter, we explore what uncertainties are and what risk means in the context of computer architecture design. We draw from other domains such as economics and financial analysis and define our own risk metric, architectural risk, to describe risk that is directly related to the performance of a computer system. We explain how it is conceived, how it is defined mathematically, as well as what the definition and metric means when put in a new intellectual analysis framework where uncertainty quantification is integrated with high level design space exploration.

With metrics and concepts defined and explained, we perform a case study at a high level in a chip-multiprocessor (CMP) setting to demonstrate that even with extremely simple and conservative assumptions, this new aspect of uncertainty and risk at the system level still reveals interesting and surprising new findings and trade-offs. The complexity of how uncertainties interact in an architecture design requires a dedicated quantitative analysis, rather than a simple

"back-of-the-envelope" calculation many may believe, to fully reveal its impact.

2.1 Uncertainty and Risk in Architecture Design

Uncertainty and risk are commonly used terms in many fields including economic and financial analytics, but it is important to be clear about their meaning here. In general, *risk* is a *function* of the impact of *uncertainty* on the return of the system. In economics, uncertainty refers to uncertain events occurring in reality. Uncertain events may include hikes in the price for raw materials, emergence of a serious competitor on the market, the loss of key personnel, and so on. Each event has some impact on the system (e.g. loss of sales, failure to recover payment). These impacts are then typically unified, by some function, to a common metric. Decisions are then made with an understanding of the risk in conjunction with expected outcome. These trade-offs can lead to the development of entirely new financial instruments.

Kaplan and Garrick provide a quantitative definition of risk [22] in the form of a "set of triplets" in Equation 2.1.

$$R = \{\langle s_i, p_i, x_i \rangle\} \tag{2.1}$$

The above definition is essentially a listing of all the uncertain events (s_i) considered, the probability (p_i) of such an event occurring and the cost (x_i) or consequence each event may lead to.

In computer architecture our default metric is performance (or some combination of performance and energy). We typically talk about the expected performance of a design without discussing the tail of the distribution of performance. Risk here maps one-to-one with the economic notion of risk in Equation 2.1: we have uncertainty in what we know; those uncertainties manifest as changes in the performance of the system; the impact of those changes can then be

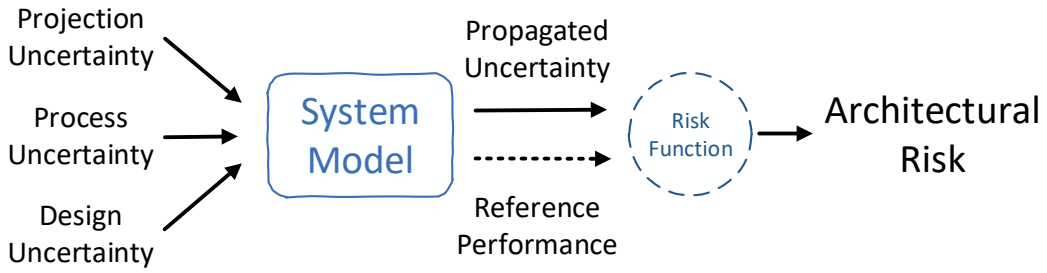


Figure 2.1: Relationship between uncertainty and architectural risk.

quantified as risk. Figure 2.1 captures this idea graphically.

For example, in Amdahl's Law, the "return" is the speedup over performance of unit core. The inputs are f, which is the parallelizable portion of the program, and s, which is the speedup obtained from the parallelization. When considering uncertainty in this case, the "uncertain events" can be *unexpected* values in f and s. The probability of these unexpected events can be modeled by some underlying distribution. The cost in terms of performance variations associated with the unexpected input values can be depicted as some cost function[1] C.

Definition. In this thesis, we define **architectural risk** in Equation 2.2 and 2.3.

$$R_e = C(P_e, \hat{P}), \quad P_e < \hat{P} \tag{2.2}$$

$$ArchR = \frac{\sum_{e \in E_1 \times E_2 \times ... \times E_n} R_e}{||E_1 \times E_2 \times ... \times E_n||} \tag{2.3}$$

Equation 2.2 captures the architectural risk under the impact of some unexpected event e. $\hat{P}$ is the reference performance (or expected/target performance of the design). If the reference performance is always guaranteed to be achieved, there is defined to be no risk. P_e is the real

[1]The term "risk function" is used interchangeably with "cost function".

system performance under the impact of an uncertain event e. C is the cost function and is usually subjective to the observer of the system, e.g., system designer or project manager. One might be interested in the probability of any unexpected event happening and thus defining a step risk function; Another might be interested in some certain events and their impacts and thus defining a piece-wise risk function; A third may be only interested in the monetary loss due to performance difference (e.g., less performant chips may be binned and sold at lower prices) and thus defining a mapping between performance and dollars. Equation 2.3 aggregates the risk captured by Equation 2.2 and takes the average across all possible event combinations, where E_1 through E_n are the sets of unexpected events for each type i of uncertainty. The performance P in the definition is not limited to execution time but is a broad term and can be any metric, including power and energy consumption, depending on what the system model is trying to evaluate.

There might be other ways of defining architectural risk over the set of uncertain events, e.g., a max function instead of taking the average like in Equation 2.3, but we follow the economic risk definition and use arithmetic mean here as a way to capture the average impact of uncertainty on a batch of system products in the context of architecture design.

To facilitate a general intellectual framework of analysis, without a particular system type of interest like Guevara et. al. [16], we propose that computer architectures are exposed to three major sources of uncertainties: projection uncertainty, process uncertainty, and design uncertainty; most of which can be tracked at different levels of the system stack.

Projection uncertainty comes from assumptions one has to make about the future. At the application level, a system design may target a specific set of applications, but those target applications may shift or change based on our understanding of the problem or new optimization techniques. We often implicitly estimate future workload behavior with measurements of existing workloads. At the device level, systems may target underlying technologies still working their way out of the research labs. The performance of these future technologies is

predicted by their physical models and there is usually some degree of uncertainty on how well they perform.

Process uncertainty comes from the manufacturing process itself. While semiconductor manufacturing is an incredibly precise process, when the probability of any fault is integrated over billions of transistors we are left with a distribution of devices. Some will work exceedingly well, others will underperform, while still others will fail to work at all. However, unlike projection uncertainty, under process uncertainty each chip is a new "roll of the dice".

Design uncertainty comes from the hardware design process itself. Components (e.g. cores and accelerators) with unresolved critical errors or introducing significant security vulnerabilities may be prevented (through a variety of means) from being accessible in an initial roll outs of a product. This class of uncertainty is a growing concern in the more heterogeneous and accelerator-dominated architectural design regime we are now faced with, and mechanisms for "partitioning out" features is an increasingly common practice.

Note that there is a *philosophical* distinction between the uncertainties we discussed above (leading to architectural risk) and the *inaccuracy* of an analytic model or simulator of a real system. It is possible to reduce the measurement inaccuracy with better modeling, more comprehensive workloads, increasingly detailed simulation, etc. — but in the end measurement inaccuracy *could* be eliminated given enough resources. However, even if one had infinite resources, the uncertainties we proposed above will still exist. It is much harder (or even impossible) to remove such uncertainties without a fundamentally new understanding of the future[2]. From a more practical standpoint, measurement error and the techniques one may use to reduce it [23–25], might either be grouped in with projection error or stand alone as a fourth category. However, we do not explore that trade-off in our work.

Importantly, as we will show later in Section 2.4, it does not take many of the above interacting forms of uncertainty to make the complexity of such interactions impossible to intuit.

[2]This is the difference between what many refer as "aleatoric" and "epistemic" uncertainty.

2.2 Related Work

2.2.1 Modeling and Design Space Exploration

Analytical models, especially closed-form analytical models, have been long used in early stage for architecture design space exploration. Probably the best known is Hill and Marty's extended Amdahl's Law for multi-core scenarios [18]. Altaf and Wood apply a similar analytical modeling to estimate performance of system with accelerators [26]. Esmaeilzadeh et. al. explore the power limit on mutli-core scaling using an extension of Hill and Marty's model [27]. Recently, Hill and Reddi explore mobile SoC design space with roofline models [28]. This line of analytical research has also led to many other works [29–32] focusing on different aspects of the system under different assumptions/simplifications.

In addition to analytical models built purely from a high-level inspection of the system, there is also a class of research using empirical modeling which exploits statistical and machine learning techniques to examine uncertainty for the purpose of inferring a better system model [23, 25, 33–43]. Many of these techniques begin with parametrized models and then statistically fit the parameters. Alameldeen and Wood in particular examine the variability in multi-threaded workloads [15] and, by injecting random errors into the detailed simulator, develop a method to account for this type of uncertainty in simulation. Most of the these works address the problem of discovering good models and/or focus on minimizing statistical errors from those models. Rather than proposing a new model, or reducing errors introduced by existing ones, we instead argue for a design and modeling approach that embraces *extrinsic uncertainties* and provides tools for making good decisions in the face of the *associated risks*. Although we examine primarily an extension to the Hill and Marty model, our analysis and framework can also be used with more detailed or complicated models as long as they can be expressed in, or approximated by, an interacting set of closed form equations.

2.2.2 Economical Thinking and Computer Architecture

This paper is certainly not the first to be inspired by economic thinking as applied to Computer Architecture. Bornholt et. al. deal with application-visible uncertain data at the programming language and runtime level [44]. Guevara et. al. combine market mechanism and resource allocation techniques to explore datacenter architectures [45]. Zahedi and Lee use game theory to study how to better allocate hardware resource in a cloud environment [46]. Fan et. al. also use game theory to handle power management in datacenters [47]. Guevara et. al. tackle the problem of runtime variation in datacenters and propose strategies to mitigate the risk of not meeting performance target [16]. These and other papers concentrate on the application of economic reasoning to better allocate resources in the face of competing interests, rather than examining the cost of extrinsic uncertainty on the high level design. Combining these lines of work would be an interesting area for future exploration.

2.3 An Evaluation Framework with Analytical Models

To demonstrate how these concepts and risk metric and help discover new insights in early stage of an architecture design, we apply the following general process mirrored from financial analytic processes: 1). pick an example architecture, 2). model it, 3). capture source uncertainties, 4). inject and propagate uncertainties through the system model, and 5). evaluate the architectural risk that it is exposed to.

2.3.1 An Example System Under Analysis

While there are many times that architects make analytic estimates of system performance, one of the most well studied is the heterogeneous core selection problem of Hill and Marty [18] as described succinctly in Table 2.1. Under this model one chooses the best performing core

design to execute the serial code (Equation 2.7) and uses the aggregated performance of all cores to execute the parallel code (Equation 2.8). Pollack's Rule [48] is used to model core performance as a function of resources consumed (Equation 2.10). Designs are bounded by the total area/resource available on chip (Equation 2.11). In addition to these classic assumptions, we also take communication overhead among different cores into account, denoted as c in Equation 2.5, which is some fraction of the sequential workload. The amount of communication overhead is proportional to the total number of cores on chip (Equation 2.9). This overhead is extensively studied in [49] and can be setup/tear down time for the parallel computation, synchronization during parallel execution, or any other overhead introduced along with parallelization.

Although real processor design is far more complicated, we show that even this simple model is significantly confounded by the introduction of uncertainty. Uncertainty in the inputs, even here, results in surprising outcomes (as detailed more in Section 2.4) and demonstrates the importance of new techniques to support this reasoning more rigorously. Our framework is by no means limited to these sets of equations only but can be applied to evaluations of different architectures including accelerators [26], different optimization objectives like power efficiency [29], or linked to more detailed simulators [33, 41]. As more parameters are added and more complex models are required this should make our proposed approach strictly more valuable and intuition even less reliable.

2.3.2 Uncertainties in our Example System

There are a total of five types of uncertainties that might be considered under the above model of a system. Uncertainties in target application behavior impact f and c (a future application running on the system might have a different level of parallelism and/or a different unit communication overhead than the benchmarks used to measure the system performance during

Table 2.1: Closed form model of performance.

$$Speedup = \frac{1}{T_{sequential} + T_{parallel}} \tag{2.4}$$

$$T_{sequential} = \frac{1 - f + c \times N_{core}}{P_{serial}} \tag{2.5}$$

$$T_{parallel} = \frac{f}{P_{parallel}} \tag{2.6}$$

$$P_{serial} = \max\{P_{core_i} \mid N_{core_i} > 0\} \tag{2.7}$$

$$P_{parallel} = \sum_{i \in core\_types} N_{core_i} \times P_{core_i} \tag{2.8}$$

$$N_{core} = \sum_{i \in core\_types} N_{core_i} \tag{2.9}$$

$$P_{core_i} = \sqrt{A_{core_i}} \tag{2.10}$$

$$A_{total} = \sum_{i \in core\_types} N_{core_i} \times A_{core_i} \tag{2.11}$$

design). Uncertainties in process/manufacturing can affect both P_{core_i} (different core instances may end up with varying performance properties due to intra-die variation) and N_{core_i} (due to fabrication defects impacting yield). Uncertainties in design may also have an effect on P_{core_i} in that, upon a design bug or failure, cores of that design might not work at all.

Each of these uncertainties is a complex thing to understand. For a technique to be useful it must *not* be highly sensitive to the assumptions about the distributions governing these unknowns. Often times we may have only a few tens of data points from which we can *infer* an underlying distribution. Later, in Section 2.3.3 we will describe exactly how this can be done in an automated way using a reversal of the classic power transform, but to evaluate the effectiveness of this approach we need *hidden reference models* to serve as a ground truth.

Pulling from the extensive literature on variation, yield, and program behavior, Table 4.2 summaries the hidden "ground truth". Our technique will attempt to capture the important aspects of these analytically from a few samples and no knowledge of the equations themselves. For N_{core_i}, i.e. the number of cores that are actually working, from its physical defini-

Table 2.2: Hidden uncertainty models.

$$f \sim \frac{Binomial(M, p)}{M} \tag{2.12}$$

$$c \sim \frac{Binomial(M, p)}{M} \tag{2.13}$$

$$N_{core_i} \sim Binomial(M, yield_{core_i}) \tag{2.14}$$

$$P_{core_i} \sim Bernoulli(p) \times LogNormal(\mu, \sigma) \tag{2.15}$$

$$yield_{core_i} = (1 + \frac{d \times A_{core_i}}{\alpha})^{-\alpha} \tag{2.16}$$

tion, we model it by a binomial distribution ranging $[0, N]$. N is the designed number of $core_i$ but each with only some probability of functioning properly taking after chip yield rate [50] (Equation 2.14, 2.16). P_{core_i} is modelled by the product of a LogNormal distribution ranging $(0, \infty]$ and a Bernoulli distribution with probability p of taking value 1 (Equation 2.15). We model core performance in such a way that it is exposed to two types of uncertainty: design uncertainty and fabrication uncertainty. Although design uncertainties can result in many consequences from degradation of performance to reduction of reliability and so on [51], we only consider severe design bugs that will lead to complete post-silicon failure of the component here. This type of uncertainty naturally follows a Bernoulli distribution by its definition, i.e. the component is either working or not. The probability of failure is set based on reported statistics [52]. Fabrication process uncertainty is modeled by the LogNormal part of the distribution. When the design works, the actual performance of each core also varies as a result of the fabrication process, leaving a Gaussian-like distribution on the positive domain [53–55]. For the LogNormal part, the location μ and scale σ is computed such that the mean performance follows Pollack's Rule (Equation 2.10) and the variance meets our desired level in experiments. For f and c, we use a normalized binomial distribution to model their uncertainties (Equation 2.12, 2.13). This distribution fits well to the characterizing data for the PARSEC benchmarks [56]. Their range is bound to be $[0, 1]$, while p is set to the mean value. M, which

is needed to construct the Binomial distribution, is computed to satisfy the level of variance we desire in simulation.

At a high level our technique requires two inputs. First, an executable architecture model under analysis (that relates a set of mutually dependent parameters capturing constraints and dependent variables to optimize). Second, a set of data points drawn from the distribution whose uncertainty you wish to consider (e.g. a set of points relating core resources and performance). If we can pull a few points from the distributions governing these models and then, without any knowledge of the model itself (just the values of the specific samples drawn from it), construct a new model that has close to the same optimization utility as the ground truth, it gives us confidence that this will be useful when applied to a specific set of trade secret data by a manufacturer[3]. If the architecture model can be described as a set of mutually dependent closed-form functions and the uncertainty in distributional representations (e.g. a large set of samples or sampling functions), our tool can symbolically combine and partially solve the closed-form equations. From this form, it can then inject and propagate uncertainty through to the final responsive metrics so that risk can be calculated. First, however, we need a way to extract (or approximate) such distributional representations of architecture uncertainty from a few initial samples.

2.3.3 Architecture Uncertainty Model Extraction

Such approximation is done by a two-phase method shown in Figure 2.2. We first test if the data set can be transformed to normality through the Box-Cox testing [57] in Step ①. If it cannot pass the test (a rare case in practice), we apply Kernel Density Estimation (KDE) methods [58] directly to the data set. These methods find a best-fit non-parametric distribution in

[3]Of course with arbitrarily complex "ground truth" such a trick is impossible. In the most general case this problem reduces to one of function inversion, which we know from cryptography can be hard, but luckily the distributions that typically govern the physical and program properties an architect would actually care about are ones that we find are highly amenable to this technique.

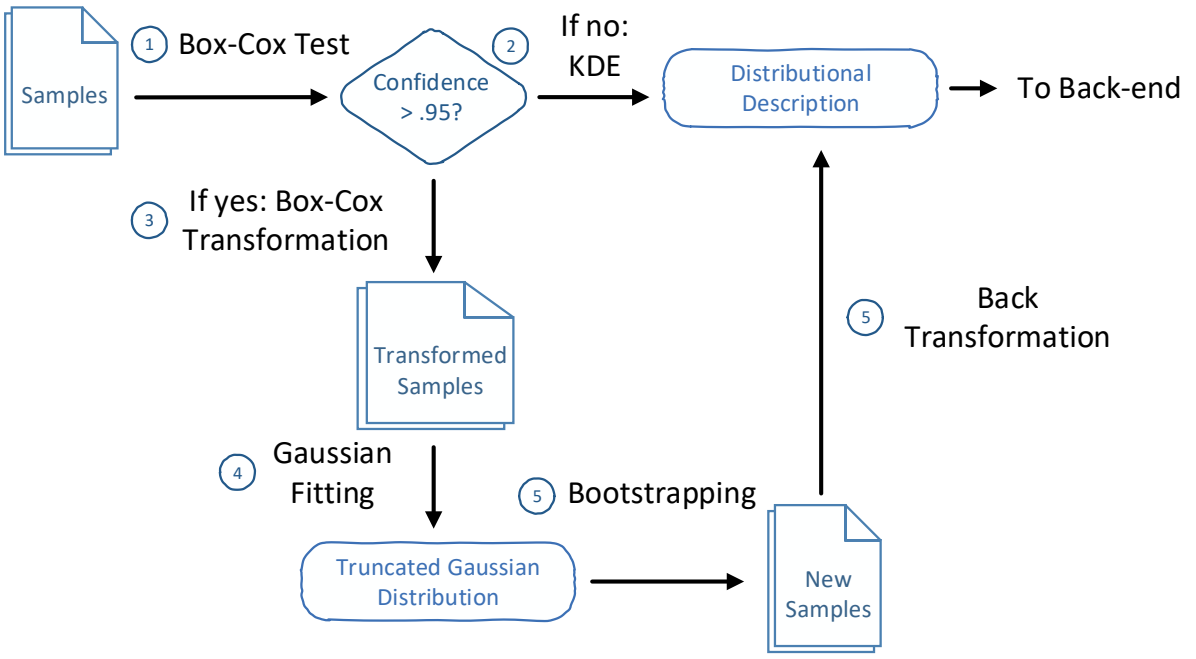


Figure 2.2: Uncertainty modeling.

Step ② and use its sampling function to facilitate uncertainty propagation. Otherwise, we transform the data set to normality using Box-Cox transformation in Step ③, re-sample (bootstrapping) from the Gaussian distribution in the transformed domain in Step ④, and back-transform the samples to the original domain. Finally we reconstruct the distribution in original domain in Step ⑤ to approximate the *hidden ground truth*. Although not as accurate as the best-fit non-parametric KDE, such bootstrapping method enables us to hand tune the desired uncertainty level in each variable and hence be able to explore the trend as input uncertainties scale. We use the bootstrapping method in our experiment to study the scaling behavior and to examine the accuracy of such approximation, but in practice, a non-parametric distribution is at most times sufficient to facilitate accurate uncertainty analysis.

Figure 2.3 gives an example of the bootstrapping process. Figure 2.3a shows the histogram generated from initial samples (the samples are taken from a log normal distribution). Figure 2.3b shows the histogram after transformation and the fitted Gaussian distribution in the

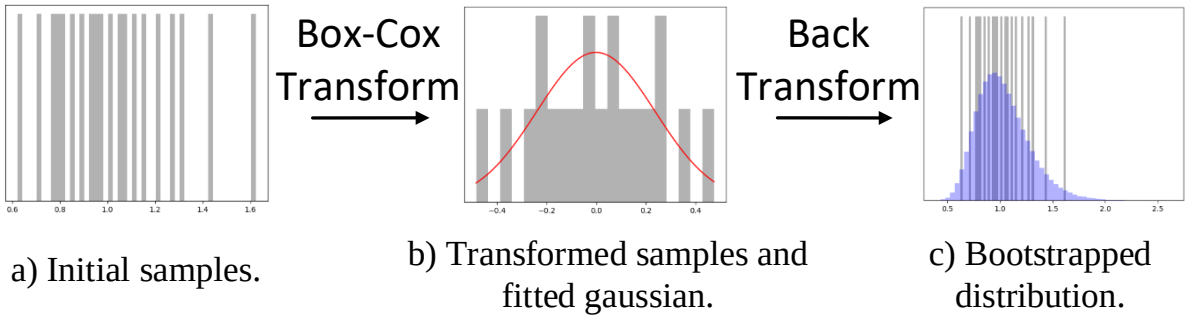


Figure 2.3: An example bootstrapping process.

transformed domain. If the initial samples can pass box-cox test, we are guaranteed to find a Gaussian distribution that fits the transformed samples. Figure 2.3c shows the bootstrapped distribution after back transformation to the original domain laid on top of the original samples.

The output of this uncertainty modeling phase is a set of uncertain variables along with their distributional descriptions to facilitate uncertainty injection and propagation.

2.3.4 Model Transformation and Execution

In Figure 2.4, we present an overview with a simple example of how the front-end modeling and symbolic execution works.

The first step ① is system modeling which builds mutually dependent equations described in Section 2.3.1. The framework then performs the following operations. ② Each variable including uncertain ones is then treated as a symbolic entity and the plain string-formatted equations are passed to symbolic execution [59]. The result is a set of algebraic equivalent equations with each symbol sitting on the left-hand side in one equation. We break the solving into steps and only resolve variables that are not uncertain, and any uncertain variable on the right-hand side in the equations is kept unresolved in its original form. This is to support uncertainty injection and propagation later. ③ The partially solved equations are then converted

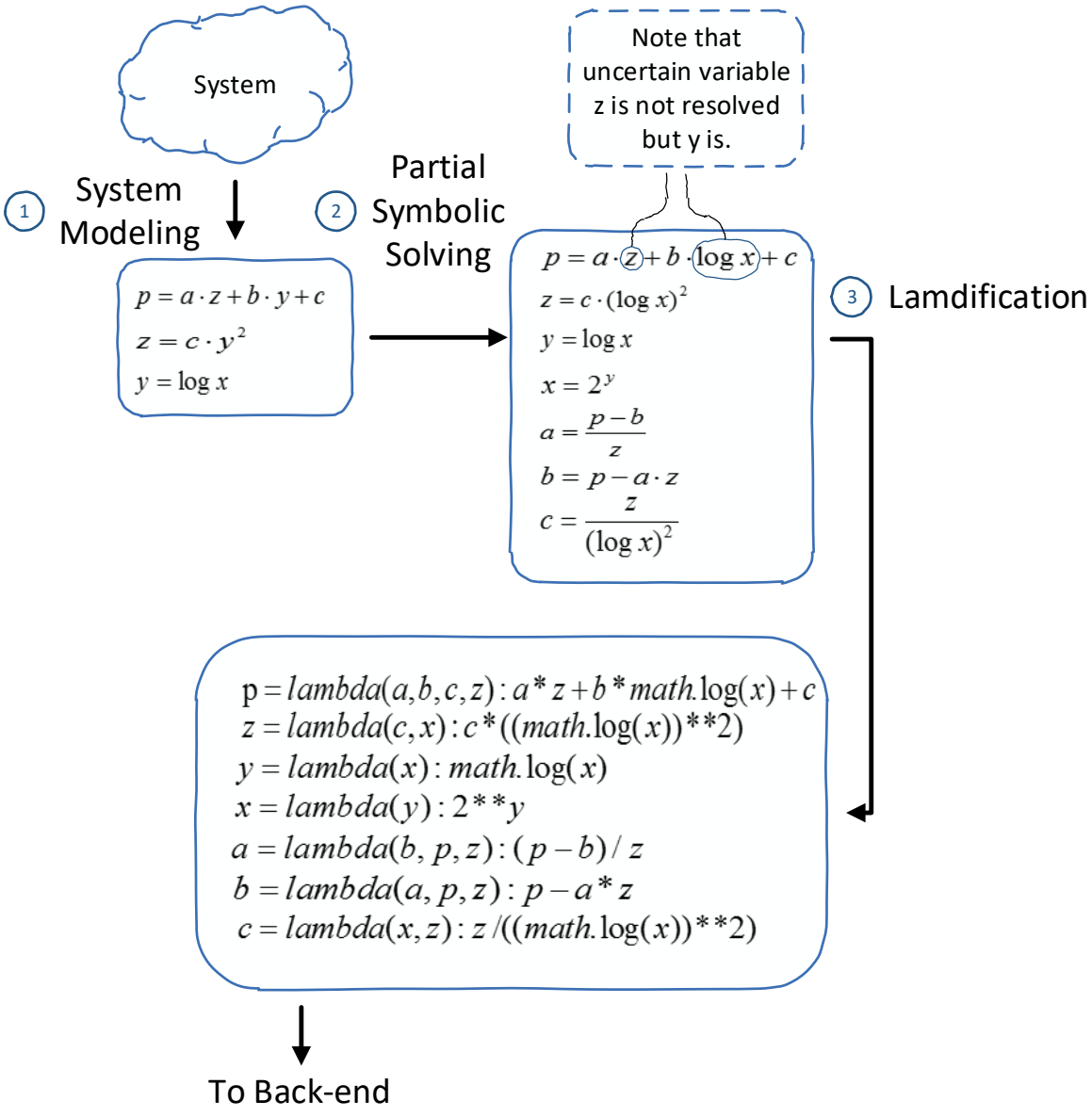


Figure 2.4: System modeling and symbolic execution.

into callable lambda functions. We also enforce a fixed argument ordering in the lamdification process.

At the end of this process we have a set of callable functions that are provided to the back-end for numerical computation.

2.3.5 Uncertainty Injection and Propagation

Figure 2.5 shows the back-end uncertainty injection and propagation process.

Given the set of functions and uncertain variables, the framework back-end then proceeds with the following steps. ① Each uncertain variable gets evaluated first as long as there are no uncertain variables in the right-hand side of their solutions. The values of certain inputs, like c and x in this case, should be provided by the system designer. ② The framework then generates distributions for all the uncertain variables. Based on their descriptions, the distribution can be generated alone or together with the evaluated value. In this example, the distribution of Z takes the evaluated value z_0 as its mean. ③ All uncertain variables appear in the argument list of the lambda functions are then replaced by the corresponding distributions. At the completion of this stage we have injected the desired uncertainties into the solved system model. ④ Each function containing distributions or random variables in the argument list is then evaluated by Latin-hypercube Monte Carlo simulation N times [60]. The result of the Monte Carlo simulation is a set of values for each responsive variable in the system. ⑤ We then re-construct a distribution from the set of values for each responsive variable. At this point, the uncertainties in the inputs are propagated into the distribution properties of the responsive variables. There are systematic errors associated with the Monte Carlo simulation but, in our experiments, we keep N sufficiently large (we use $N = 10,000$) to keep the errors negligible. ⑥ Finally, we calculate risk based on the distribution of the responsive variable and the risk function provided.

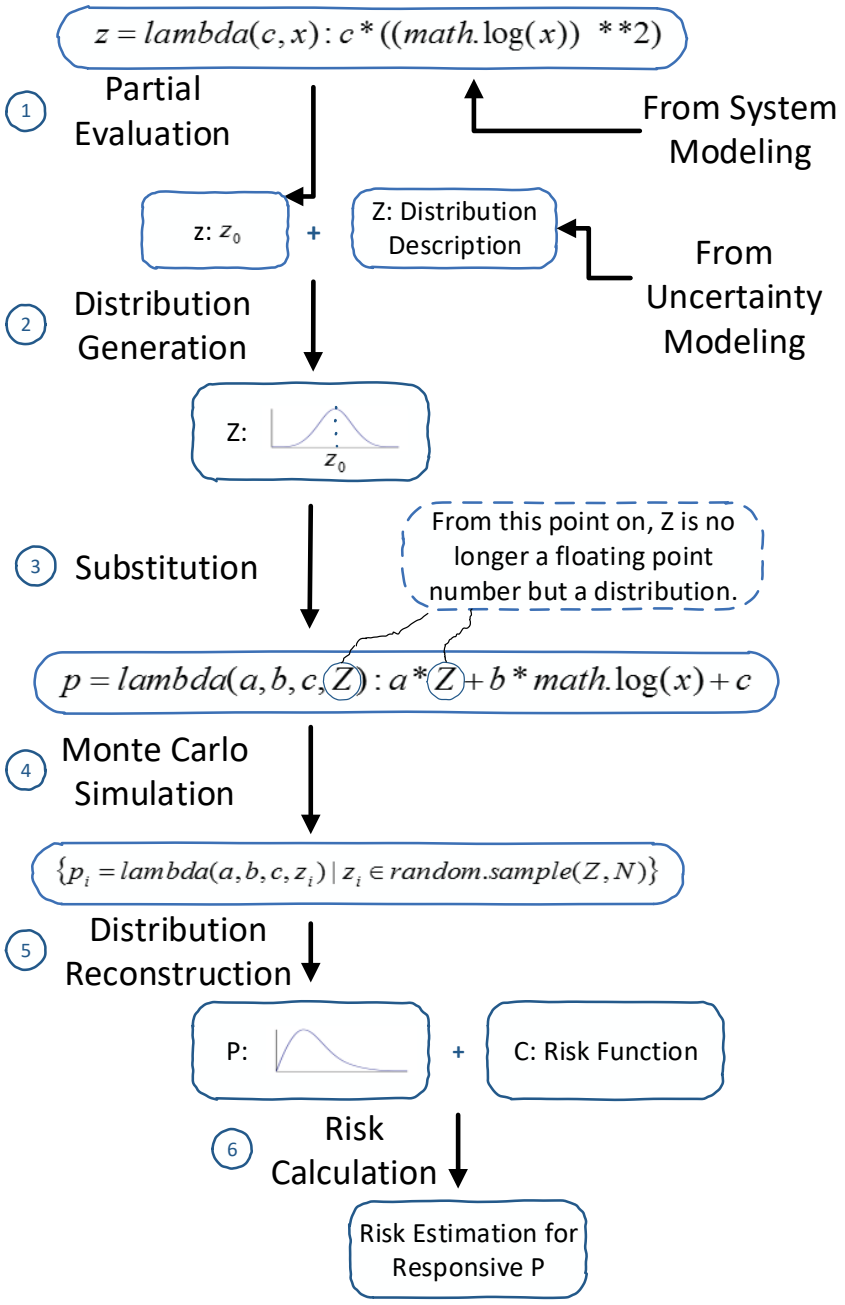


Figure 2.5: Back-end uncertainty injection and propagation.

2.4 Exploring The Design Space under Uncertainties and Risk

Building from the framework and ground truth models discussed above, we now carry out an analysis on the classic Hill and Marty core selection design problem and demonstrate how performance and risk interrelate and can even be co-managed. The design question is essentially: what cores and how many of them should we put on a CMP in the face of uncertainties? We bound the design space to populate by constraining the total chip size (or resources) to be 256 units and consider the full spectrum of designs (rather than just one big-core coupled with many tiny cores). Specifically we ask the following questions which will be answered by a series of implications we draw from our experiment results:

- How does uncertainty manifest and interact in the system?
- How sensitive is CMP performance in the face of uncertainty?
- Is the conventional risk-oblivious design optimal in terms of architectural risk? Furthermore, is it still optimal even in terms of expected performance?
- When is there a trade-off space between architectural risk and expected performance? What does the trade-off space look like?
- What configuration/design is favored when one considers risk?

After exploring the design space with ground truth distributions, we show that our approximation method still leads to optimal or near-optimal designs from only a handful of samples. Such partial information about the underlying uncertainty distributions is often the case in early modeling and design cycles. While we primarily consider architectural risk in the form of performance, we further demonstrate the use of this analysis in evaluating *monetary* risk function using both ground truth distributions and the approximations.

2.4.1 Uncertainty Manifestation

Experiment Setup. In order to answer the question of how uncertainty manifests in the output, we inject a total of five types of uncertainties into the four input variables of our model. The uncertainties we inject are application characteristics uncertainties in f and c, process variation in P_{core_i} and N_{core_i} as well as design uncertainty in P_{core_i}.

Table 2.3: Injected uncertainties.

Input	Certain Value	Uncertain Value	
		Mean	Std
f	$\hat{f}$	$\hat{f}$	$\sigma \cdot (1 - \hat{f})$
c	$\hat{c}$	$\hat{c}$	$\sigma \cdot \hat{c}$
P_{core_i}	$\hat{P}_i$	$\hat{P}_i$	$\sigma \cdot \hat{P}_i$
Fabric	$N_{core_i} \sim Binomial(\hat{N}_{core_i}, \quad yield_{core_i})$		
Design	$P_{core_i} \sim Bernoulli(\sigma \cdot \gamma) \times P_{core_i}$		

Table 2.3 describes how much uncertainty we inject. With $\sigma = 0$, the inputs are certain values ($\hat{f}$, $\hat{c}$, $\hat{P}_{core_i}$ and $\hat{N}_{core_i}$), the resulting performance is the conventional "certain" result without propagated uncertainty. With $\sigma > 0$, f is centered on $\hat{f}$ with a standard deviation of $\sigma \cdot (1 - \hat{f})$, such that the standard deviation of f is kept small enough that f only varies at the least significant digit of $\hat{f}$ and does not completely change the application to another category (again, we are being conservative here and a larger uncertainty will make the risk even more important). Similarly, c is centered on $\hat{c}$ but with a standard deviation of $\sigma \cdot \hat{c}$ as c in itself is very close to 0. As for core performance P_{core_i}, the performance uncertainty is centered around $\hat{P}_{core_i}$ and the standard deviation is set to be $\sigma \cdot \hat{P}_{core_i}$. There are two special types of uncertainty that are not centered around the corresponding certain values. The fabrication uncertainty is added when we consider that each core has a probability of failure (not functioning properly). We keep yield rate for each type of core evaluated constant throughout the computation. Table 2.4

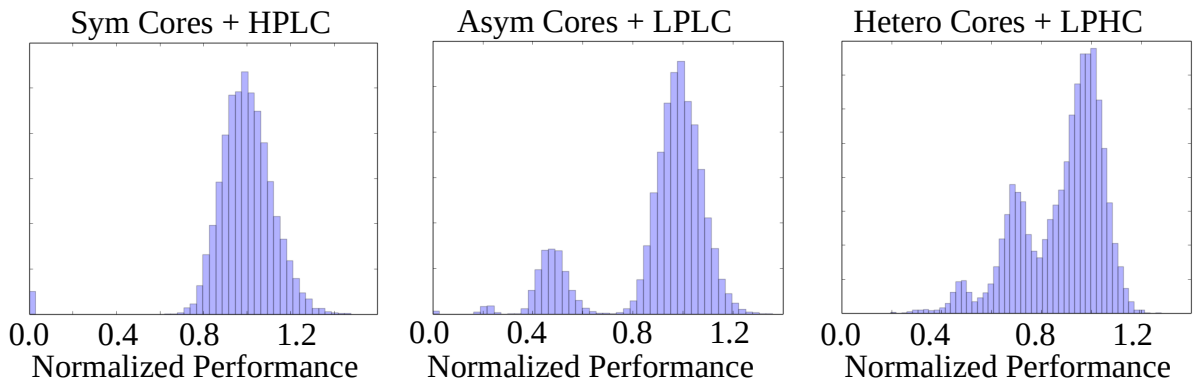


Figure 2.6: Performance distribution under uncertainties. "Sym Cores" stands for a configuration of 32x8 (32 small cores of size 8), "Asym Cores" stands for a configuration of 1x128 + 16x8 (one large core of size 128 and 16 small cores of size 8), and "Hetero Cores" stands for a full heterogeneous configuration of 2x8 + 1x16 + 1x32 + 1x64 + 1x128.

lists the yield rates computed using Poisson chip yield model [61]. Note that yield rate is not dependent on σ but only on core size. Design uncertainty is modeled by a Bernoulli with probability $\sigma \cdot \gamma$, and we set the intrinsic probability γ based on an estimation of existing data [52].

Table 2.4: Yield rates.

core size	8	16	32	64	128
yield	98%	96%	92%	85%	75%

We hand-tune the injected uncertainty level σ from 0 to 1 for four different categories of applications and three different architecture designs. The four applications are characterized by different values of parallelizable portion $\hat{f}$ and unit communication overhead $\hat{c}$. We call $\hat{f} = 0.999$ high parallelism (HP) and $\hat{f} = 0.9$ low parallelism (LP). We refer to $\hat{c} = 0.001$ as low communication cost (LC) and $\hat{c} = 0.01$ as high communication cost (HC). To see the impact we consider three example designs which are symmetric (32x8) and asymmetric (1x128 + 16x8) designs from Hill and Marty's setting as well as an extended full heterogeneous architecture in which 5 types of cores are present.

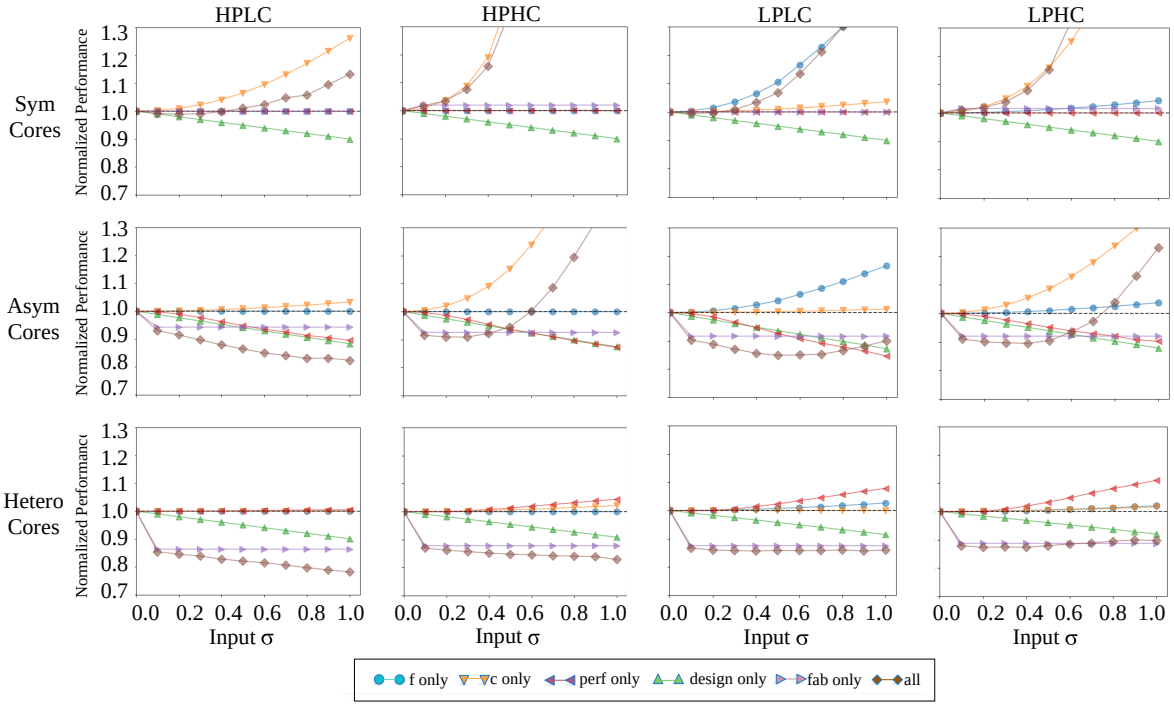


Figure 2.7: Uncertainty manifestation on output performance. Legend indicates which type of uncertainty is under consideration, and expected performance is normalized to risk-unaware performance.

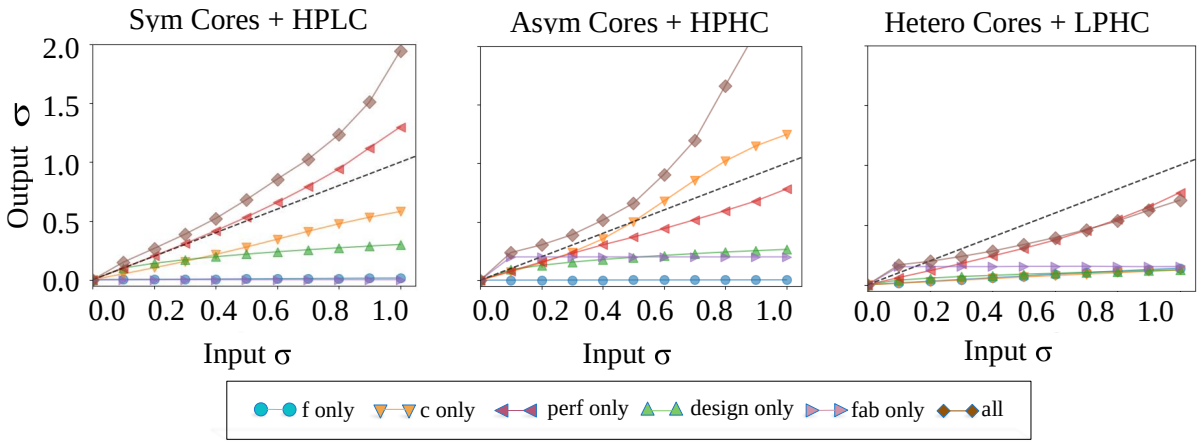


Figure 2.8: Example uncertainty manifestation on output uncertainty. Legend indicates which type of uncertainty is under consideration and standard deviation is normalized to risk-unaware performance.

Results and Discussion. In Figure 2.6, we show examples on the resulting performance distribution. The mean of the performance distribution should be the expected performance under uncertainties and its standard deviation (or variance) measures how much each chip differs from one another in terms of performance. The shape of the distribution also matters in that it relates to how much architectural risk a design is exposed to. Based on our architectural risk definition in Section 2.1, given a reference performance (or performance goal), risk is essentially a weighted area under the curve to the left of the performance goal. Looking at the figure, an important observation is that the resulting performance distribution is very irregular even with our simple and regular input distributions.

In Figure 2.7, we present how the expected performance under the impact of input uncertainties behaves as σ increases. In most cases, input uncertainties lead to worse expected performance compared to its "certain" version, while in some cases, a strong uncertainty, especially on f and c, can lead to better expected performance because of the asymmetric impact f and c have on performance. For example, in the asymmetric architecture with an application of $\hat{f} = 0.9$ and $\hat{c} = 0.001$, an $f = 0.9+0.1$ will raise the performance from 36.38 to 37.93 (an 1.55 increase) while an $f = 0.9 - 0.1$ lowers performance to 34.96 (an 1.42 decrease). In another

word, given the same deviation from $\hat{f}$, the impact of a higher f is greater than the impact of a lower f, resulting in a better expected performance. From an architectural point of view, this asymmetric impact results from the fact that asymmetric design often fits applications with more inherent parallelism better.

If we compare the performance across different applications for the symmetric design, we can see that when f gets smaller (uncertainty on f gets larger) and c unchanged, the impact of uncertainty on f becomes dominant (compare the first and third figure on the first row), while when c gets larger (uncertainty on c grows) and f unchanged, the impact of uncertainty on c becomes dominant (compare the third and fourth figure on the first row). This observation holds for all three designs and meets our expectation that the uncertainty on the dominating characteristic of the application has a greater impact on the output.

If we instead pick an application and compare across all three designs, we can tell that the performance "boost" brought by the asymmetry on f and c diminishes as the chip becomes more heterogeneous. The same example math of $f = 0.9 + 0.1$ and $f = 0.9 - 0.1$ suffices to show that the asymmetry is barely observable in the very heterogeneous design. In another word, the more heterogeneous the chip is, the less sensitive to application uncertainty it is.

Another observation begins to surface in this comparison study. The overall impact of uncertainty on core performance P_{core_i} behaves differently for chips of different heterogeneity and heterogeneous chips are generally more sensitive to architectural uncertainties. In the symmetric case, all cores are of the same size and the uncertainty in the performance of each core cancels out one another, leaving the result performance unchanged. While in the asymmetric case, the collection of the small cores still behave in such a way, but when the big core has a degraded performance, it has a much larger impact on performance that cannot be offset by the other smaller cores. In the very heterogeneous design, however, the collective behavior of the cores contributes to a better expected performance. This explains why the impact of architecture uncertainty grows when the architecture design becomes more heterogeneous.

Figure 2.8 gives three examples of the uncertainty in the output of the model as the input uncertainties vary. In general, uncertainty in performance grows as the input σ increases. This follows our intuition that the more uncertain the input is, the more uncertain the output should be. Most of the input uncertainties propagate through the model sub-linearly, indicating some tolerance for uncertainty the model exhibits. We also compare across the designs for the same application, and find out that the more heterogeneous the chip is, the more uncertainty-tolerant it is.

A counter-intuitive fact is that the composite uncertainty in the output is not simply an accumulation of all the input uncertainties. In fact, the uncertainties are not even additive. To better demonstrate this behavior, we conduct a series of experiments by removing one type of uncertainty at a time. In Figure 2.9, we use the asymmetric design as an example to show that the output uncertainty sometimes rises when there is less uncertainty in some of the inputs. This happens because uncertainty has two possible effects on the output performance: it may contribute to a better performance or it may lead to a worse performance. And different components of the system (different inputs) respond to uncertainty with different magnitudes, as well as directions (better or worse). When combined, different uncertain inputs may enhance each other, leaving performance shifted more from the expected value, while in other situations they may attenuate each other reducing shifts in performance.

In summary, regarding how uncertainties propagate and how sensitive CMPs to these input uncertainties, we have the following implications.

Implication 1. Uncertainties propagate through the model with non-intuitive interaction, the resulting performance distribution is beyond what a simple "back-of-the-envelope" estimation can reveal.

Implication 2. The more heterogeneous the chip is, the less sensitive its expected performance to application uncertainty, but the more sensitive its expected performance to architecture uncertainty.

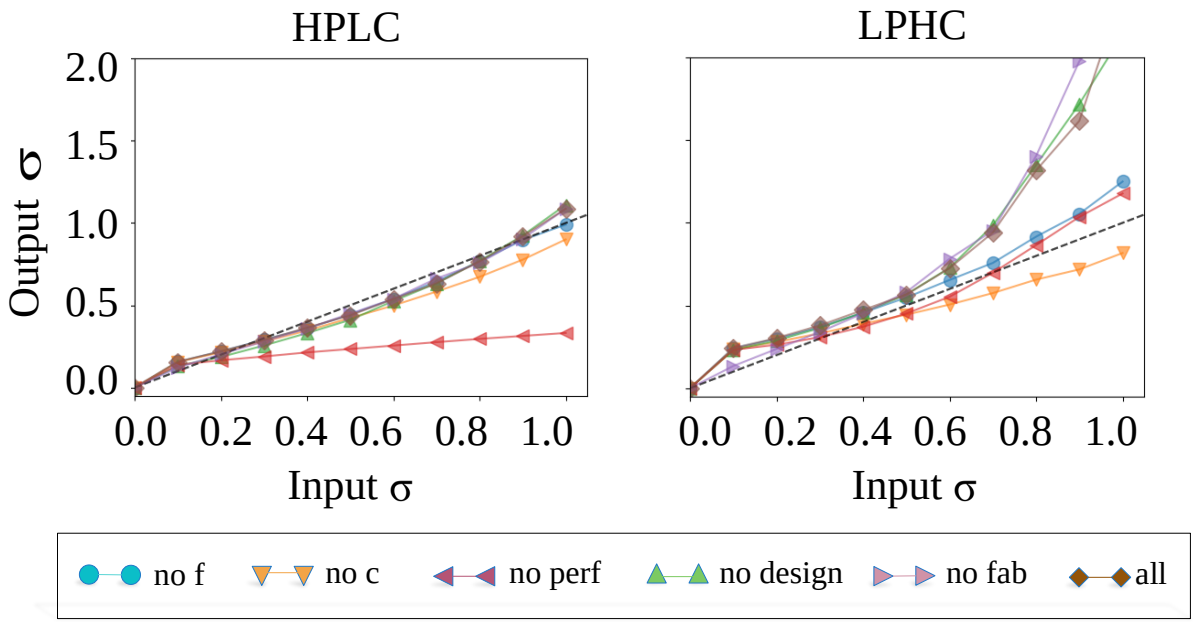


Figure 2.9: Non-accumulative output uncertainty for asymmetric architecture. Legend indicates which type of uncertainty is excluded and all means all types of uncertainty are considered.

Implication 3. The more heterogeneous the chip is, the more tolerant/robust its performance is to input uncertainties.

2.4.2 Impact on Design

Given the complexity of propagated uncertainties in the three designs above, we now expand our search space and explore how uncertainty and risk may impact design decisions in the uncertainty wrought design space.

Experiment Setup. The injected uncertainties are identical to the setups in Section 2.4.1. We exhaustively enumerate all valid design options. Each valid design option is a configuration taking up 256 on-chip resource units (a total size of 256) with a combination of different cores, each of which having a size of power of two ranging from 8 to 256 (e.g. a valid design can be 32 cores of size 8, 1 core of size 256 or 16 cores of size 8 plus 1 core of size 128). Some combination does not consume all the on-chip resources, and in those cases, we group all

resource left into one additional core (e.g. 8 cores of size 8 plus one core of size 192 is also valid). We also use a quadratic risk function in this exploration. In other words risk is the sum square of the performance loss below some reference due to uncertainty for all eventualities. The idea is that performance well below expectation is much worse than performance just below expectation (similar to minimizing sum square error).

Results and Discussion. We present results considering both architecture uncertainties (process uncertainty and design uncertainty) and application uncertainties (uncertainty in f and c) in Figure 2.10. The conventional performance-optimizing uncertainty-oblivious design is at most times not the optimal choice not only in terms of risk but also, very counter-intuitively, even in terms of expected performance. If we take a look at the architecture uncertainties (σ_arch on the y-axis) and application uncertainties (σ_app on the x-axis) respectively, we can tell that architecture uncertainties usually impose a larger impact on design decisions. In all four types of applications, the optimal design shifting along the y-axis occurs even when there is only 20% uncertainty or less. Meanwhile, the application uncertainties shift the best design at a slower pace. If we consider application uncertainty alone (with a fixed architecture uncertainty), we can see that the application uncertainties shift the risk-optimal design easily but the performance optimality shifts only when application uncertainty is abundant. In one case where parallelism is high and communication overhead is low, application uncertainty does not shift the performance optimal design at all even at a level of 100% of the mean.

An example trade-off space with LPHC application between performance-optimal design and risk-optimal design is shown in Figure 2.11. To help readability and understanding, we do not include all curves at every input uncertainty level in Figure 2.11a, but the trends of other curves are very similar to the examples we show in the figure. We can tell that the amount of input uncertainties shifts the possible outcomes of all designs in the performance-risk space and determines how the trade-off space look like. In most cases, there exists a trade-off space between the performance-optimal design and the risk-optimal design. Taking the curve marked

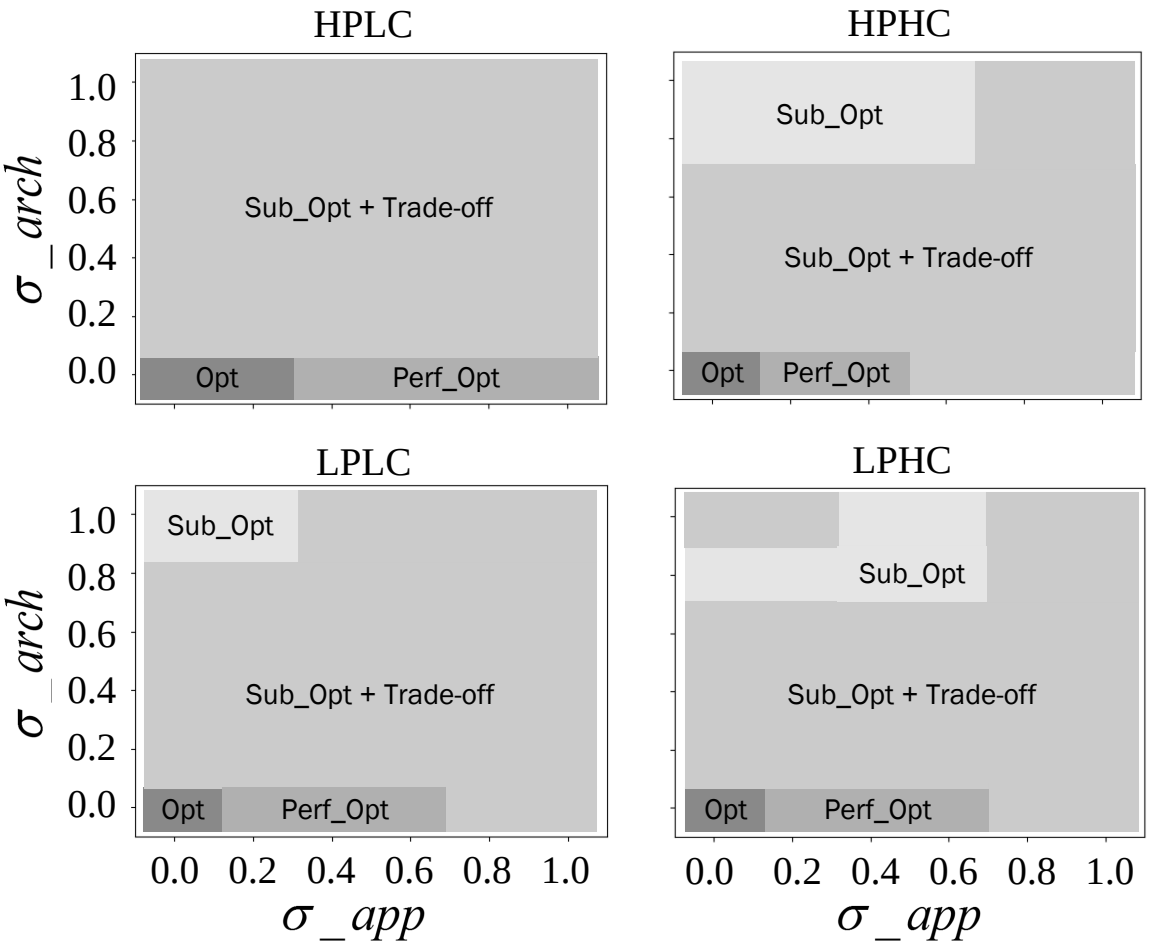
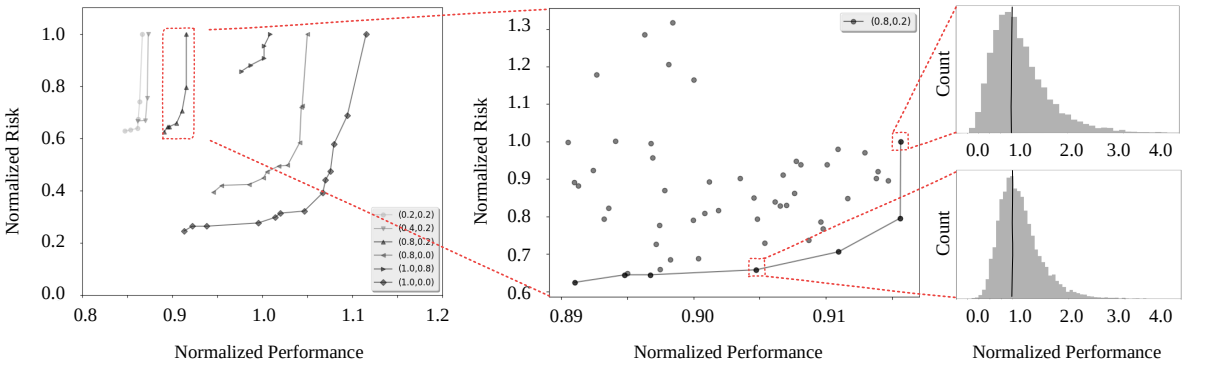


Figure 2.10: Impact on design of application uncertainties and architecture uncertainties. σ_app is the uncertainties in f and c and σ_arch is the uncertainties in P_{core_i} and N_{core_i}. "Opt" means conventional design is optimal in both expected performance and risk. "Perf Opt Only" means conventional design is only optimal in terms of expected performance. "Sub-opt" means conventional design is strictly sub-optimal in both expected performance and risk. "Sub-opt + Tradeoff" means not only conventional design is sub-optimal, there is also a trade-off space between performance-optimal design and risk-optimal design.



a) Pareto curves at eifferent uncertainty level. b) Zoom-in Pareto curve and non-Pareto points. c) Zoom-in performance distribution.

Figure 2.11: Example trade-off space between performance-optimal design and risk-optimal design. Each point in the space is a performance distribution of a certain design (core-selection) at some input uncertainty level (denoted as a tuple of (σ_app, σ_arch)) with the LPHC application. Performance is normalized to that of the conventional case (risk-oblivious performance-optimal design). Risk is normalized to that of the performance-optimal design at each input uncertainty level.

in Figure 2.11a as an example, one can mitigate almost 60% of risk at the cost of less than 3% performance. Figure 2.11b zooms in on the example curve and shows both the Pareto-optimal designs as well as the non-optimal designs with relatively strong expected performance (within 89% of the best expected performance) in the space. Figure 2.11c further zooms in on the two representative designs on the Pareto curve. We can see that having a more "concentrated" distribution around the performance goal helps bring down the architectural risk of the lower design, while the upper design has a wider distribution leaving a larger risk but better expected performance.

In summary, regarding how conventional design performs in the uncertainty wrought design space, we have the following implications.

Implication 4. Conventional architectural risk-oblivious design is at most times not the architectural risk optimal design when there's only moderate amount of input uncertainties. Conventional design is also oftentimes not even optimal in terms of expected performance, i.e. there is another core-configuration that yields better expected performance in the face of

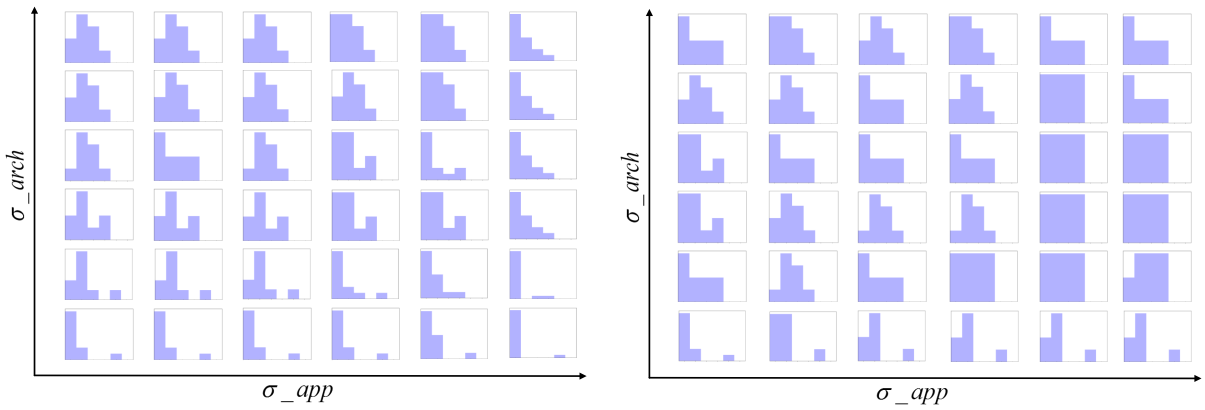
uncertainty.

Implication 5. Architecture uncertainties usually have a larger impact on design optimality while application uncertainties have a relatively smaller impact.

Implication 6. At most times there exists a trade-off space between performance-optimal design and risk-optimal design, and one can mitigate a good amount of risk at the cost of a relatively small performance degradation.

Next we explore what exactly the optimal core configurations are and what configurations are generally preferred in terms of both expected performance and architectural risk using results with LPHC as an example. Figure 2.12a gives the performance-optimal core configuration distributions in our search space. If we consider application uncertainty σ_app alone, when it gets larger, the histograms tend to grow towards the left edge and concentrate on a few selections. This "concentrating" trend means that *more asymmetric* configurations are generally favored when there is more application uncertainty. This trend results from the asymmetric impact we discussed in Section 2.4.1. A large core is needed to compensate the performance loss due to a lower f or higher c while the herd of small cores are better performing than a distribution of heterogeneous cores for parallel execution. However, when σ_arch gets larger, i.e. there are more architecture uncertainties involved, we can see that the histograms tend to spread out across different core sizes. This "spreading" trend indicates that *less asymmetric* configurations are preferred when there is more architecture uncertainty. This is due to the fact that mid-sized cores are chosen because the performances of cores have variations and a mid-sized core can step in during serial execution to compensate the performance loss due to a less performant large core. Another reason is that multiple cores of each type are chosen to fight the intra-die process variation, leading to fewer core types on chip because of total area/resource constraint. These two counter-directional trends are the main reasons that the design space is very irregular and complicated.

Similarly in Figure 2.12b, we show the optimal core configuration for architectural risk



a) Expected performance optimal designs

b) Architectural risk optimal designs

Figure 2.12: Core configurations of optimal designs for LPHC. Each design is represented in a histogram of core distribution. Each bar in the histogram corresponds to the count for each type of core of which the size ranging 8, 16, 32, 64, 128, and 256 from left to right . σ_app and σ_arch both range from 0 to 1. Note that all designs are bounded by the same area constraint and the y-axis of each histogram is not normalized to better show the ratio between different types of cores.

optimal designs. Comparing with Figure 2.12a, we can tell that in most cases a "spread-out" or *more symmetric* configuration is needed to minimize architectural risk. However, the general trend when uncertainty grows is a blend of such "spreading" and those in Figure 2.12a and is very irregular and extremely hard, if not impossible, to intuit.

To sum up, regarding what configuration/design is in general more favorable, we have the following implications.

Implication 7. More symmetric configurations tend to minimize architectural risk while more asymmetric configurations tend to maximize expected performance in the face of uncertainties.

Implication 8. As application uncertainty gets larger, more asymmetric configurations are preferred, while as architecture uncertainty gets larger, more symmetric configurations are favored.

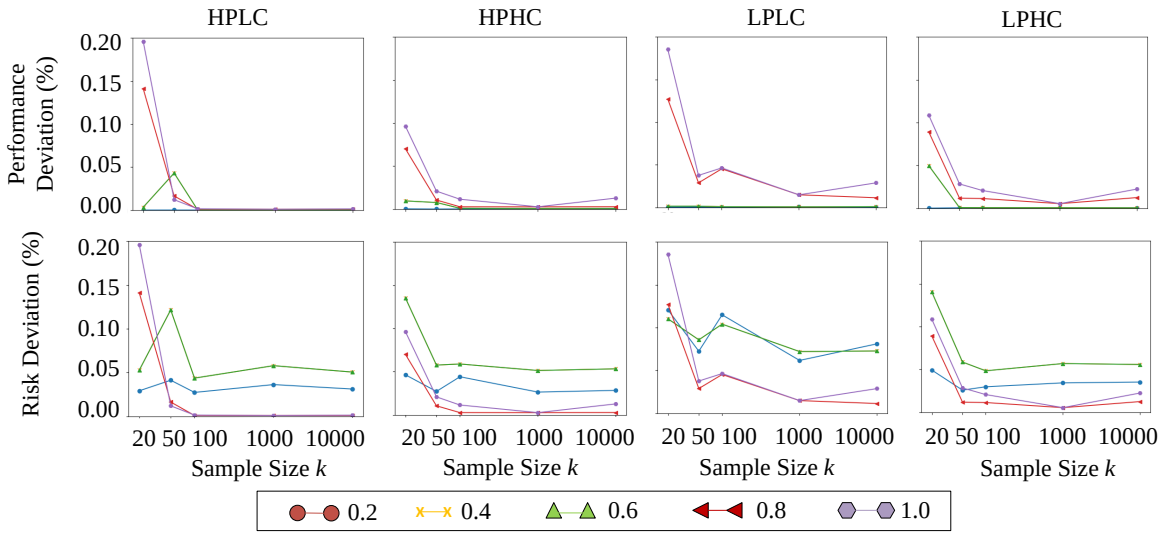


Figure 2.13: Quality of approximation. Different curves are averages taken for input uncertainties (both σ_app and σ_arch) less than a given threshold marked by corresponding legend.

2.4.3 Approximating Uncertainty and Risk

In this section, we explore, when there is no a prior knowledge of the *hidden ground truth* distributions of the input uncertainties but merely a few samples drawn from them, how our approximation method performs.

Experiment Setup. We take only k samples from each of the distributional input uncertainties listed in Table 4.2. We then apply our bootstrapping method discussed in Section 2.3.3 on these samples to acquire approximations to the true uncertainty distributions. We then again conduct an exhaustive search through the design space using settings in Section 2.4.2 with the approximate uncertainties.

Results and Discussion. Figure 2.13 presents the approximation quality. Aside from some numerical fluctuations, the general trend is clearly showing that we can bound the error in terms of both expected performance and risk within 5% with 50 samples or even fewer. When the sample size exceeds 100, the quality of approximation is very stable and the approximation is very close to the hidden ground truth. We will further show the approximation quality with a

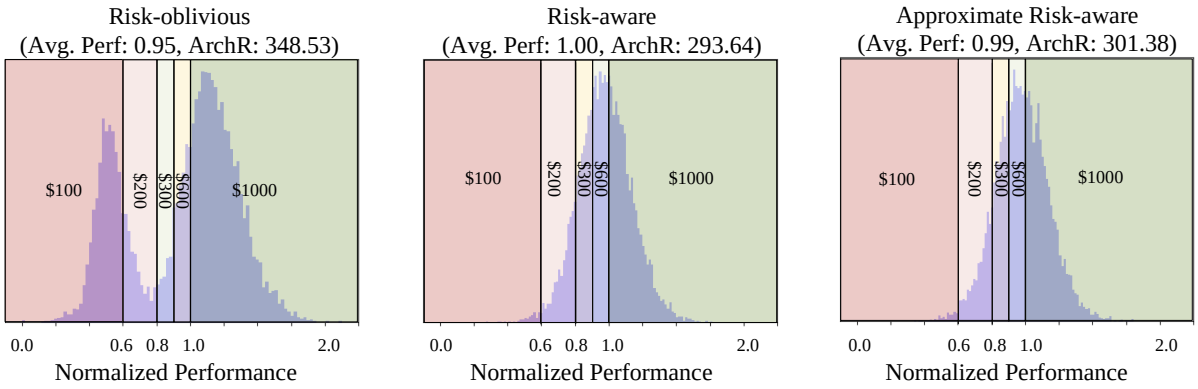


Figure 2.14: Binning of design results under uncertainties. A price binning based on Table 2.5 is laid on top of the performance distribution derived from each design with LPHC application at an input uncertainty of $\sigma\_app = \sigma\_arch = 0.2$. These designs correspond to the point in Figure 2.10 LPHC at coordinate (0.2, 0.2).

concrete example in Section 2.4.4.

2.4.4 From Architectural Risk to Financial Risk

Now that we understand the impact of uncertainty and risk on core selection, we take a step back and re-examine the meaning of our risk function. The quadratic risk function used in above experiments may seem a little abstract when it comes to interpreting the meaning of its absolute value. One may ask the question of: *what is the cost in dollars if a sub-optimal design is chosen in the face of uncertainty?*

To answer this type of question, a different risk function which ties performance to a concrete dollar value in the market is needed. Here we show an example using a simple monetary mapping for normalized chip performance. Table 2.5 lists the relationship of normalized performance and dollars estimated from a publicly available CPU price list [62].

Table 2.5: Correlation between Normalized Chip Performance and Market Value.

Perf	¡ 0.6	[0.6, 0.8)	[0.8, 0.9)	[0.9, 1.0)	≥ 1.0
$	100	200	300	600	1000

The risk function is defined in Equation 2.17 to reveal risk in terms of dollar cost due to performance uncertainty. We consider all input uncertainties at $\sigma = 0.2$ with LPHC as an example and derive the performance distribution of a risk-unaware optimal design, a risk-aware optimal design with *hidden ground truth*, and an approximate risk-aware optimal design with sample size $k = 50$ in Figure 2.14. Based on our definition in Section 2.1, the architectural risk of the conventional design is \$348.53 which means \$348.53 per chip on average lost (compared to the price of a chip at performance 1.0) due to uncertainty with an average case performance of 0.95. For risk-aware optimal design with hidden ground truth, the architecture risk is \$293.64 with an average performance of 1.00. For the approximated risk-aware optimal design, the architectural risk is \$301.38 with an expected performance of 0.99. At this point, for this specific setting, we can answer that *\$47.15 per chip can be saved with even better expected chip performance using an approximate uncertainty analysis.*

$$C(P_e, \hat{P}) = \$(\hat{P}) - \$(P_e) \tag{2.17}$$

2.5 Chapter Summary

We are living in interesting times for computer architecture — both from above by the applications, and below by the technology, we find ourselves pressed between many new uncertainties. While developing new computer system has always involved a risk of failing to meet performance goals, the new magnitude of these uncertainties may now lead to either overly conservative design practices on one hand, or "fragile" designs on the other. The *degree* to which uncertainty actually changes the expected performance of a design (and thus the nature of what an "optimal" design really is) is not something that has been discussed much in prior work. In this chapter we show that it is possible to define, model, and quantify architectural

risk.

Luckily the ideas of risk and risk-management are well understood in economics and by drawing upon this expertise, we are able to describe a new analytic framework for high-level risk-aware architectural analysis. We show that ignoring the degree to which parameters are unknown, even under fairly simple and conservative performance modeling assumptions, can lead to designs with radically different risk profiles. While there is always a design that maximizes expected performance, we show in this chapter that, even absent the confounding factors of cost, energy, thermal constraints, etc., the "optimal" design may only be a point in the risk-performance trade-off space.

Chapter 3

Supporting Analytical Modeling in Architecture Designs

We have shown an example high-level architecture analysis in Chapter 2, yet such an analysis in reality is most likely an ad-hoc practice less structured than the approach we take. The process described in Chapter 2, although more organized than just a set of python scripts or spreadsheets, is still far from being systematic enough that it is easy to use and build upon. Here, we start from reviewing the pain points of engineering such analysis in practice. These pitfalls and drawbacks motivate us to propose a more structured and systematic way of managing these high-level architecture models and analysis in this Chapter. We describe Charm, a closed-form high-level architecture modeling language. Through a complete example code, we discuss the principles that Charm is designed upon and how it addresses the pain points with ad-hoc methods. We then elaborate how the language, as well as the interpretation process, are designed and implemented. Finally, we perform three case studies to demonstrate the capabilities and benefits of performing such high-level analysis with Charm.

3.1 Understanding Pain Points of Ad-hoc Analysis

To understand Charm it is useful to have a running example. In this section, we present an implementation of the model and analysis from a well-cited study of dark silicon scaling [63]. After a brief review of the models, we show the complete code in Charm performing the same analysis of symmetric topology with ITRS technology scaling predictions. As we extend this model to cover more analysis provided in [63], it leads to a discussion of the potential issues with less structured approaches and highlights some of the features of the language that help architects avoid these pitfalls.

3.1.1 A Brief Review of the Dark Silicon Model

To forecast the degree to which dark silicon will become prevelent on CMPs under process scaling, Esmaeilzadeh et al. first construct three models: a device model (*DevM*), a core model (*CorM*) and a CMP model (*CmpM*). *DevM* is the technology scaling model relating *tech node* to *frequency scaling factor* and *power scaling factor*. It is a composite model combining a scaling prediction with a simple dynamic power model ($P = \alpha C V_{dd}^2 f$). *CorM* is the model relating *core performance*, *core power*, and *core area*. It is empirically deduced by fitting real processor data points. *CmpM* has two flavors which are essentially very different models: $CmpM_U$ and $CmpM_R$. $CmpM_U$ is an extension of the Hill-Marty CMP model [64] and $CmpM_R$ is a mechanistic model [65].

A composition of the three models is then used to drive the design space exploration. The authors combine *DevM* and *CorM* to look at *CorM* for different *tech node* and combine *DevM*, *CorM*, and *CmpM* to iterate over a collections of topologies, scaling predictions and core configurations. They then plot the scaling curves for the dynamic topology/$CmpM_R$ with both ITRS and a conservative scaling [66].

3.1.2 A Complete Charm Code Example

Listing 3.1 gives the complete code in Charm DSL to run the design space exploration with ITRS predictions on the symmetric topology (we later extend the analysis to other topologies and predictions in Section 3.5.1). At a high level, we can see that the code is split into three major components: type definition (Line 3-8[1]), model specification (Line 11-56) and analysis declaration (Line 59-66).

Specifically, we first define commonly used domains as Charm types on the architectural quantities we care about (Line 3-8). For example, the parallelism parameter in the model has a physical meaning of the proportion of the algorithm that can be parallelized and it naturally falls between $[0, 1]$. We thus define a type $Fraction$ to encapsulate this domain constraint. While this is a simple example, more complex constraints are possible.

We then formally specify the three models ($DevM$, $CorM$, $CmpM$) to evaluate (Line 11-56). Taking the *ExtendedPollacksRule* model (Line 34-41) as an example, it declares upfront all the architectural quantities that are involved in the model (e.g., *ref_core_area* which is the core size at the reference technology node), their types (e.g., *ref_core_area* is a real number on the positive domain) and the relationships between the architectural quantities, e.g., $area = 0.0152perf^2 + 0.0265perf + 7.4393$ (the constants come directly from the original dark silicon paper [63]).

Once the models are defined, it is straightforward to declare the analysis in Charm (Line 59-66). One simply selects the given models in the study, supplies the inputs and specifies the target metrics to explore. For example, in this case, we select *ITRS*, *ExtendedPollacksRule* and *SymmetricAmdahl* models (Line 59), provide values such as the area (Line 60) and power (Line 61) constraints, and finally tell Charm what quantities we care to explore, in this case *speedup*, *dark_silicon_ratio* and *core_num* (Line 66).

[1]Line numbers in Section 3.1.2 all refer to Listing 3.1 and Listing 3.2 unless otherwise specified.

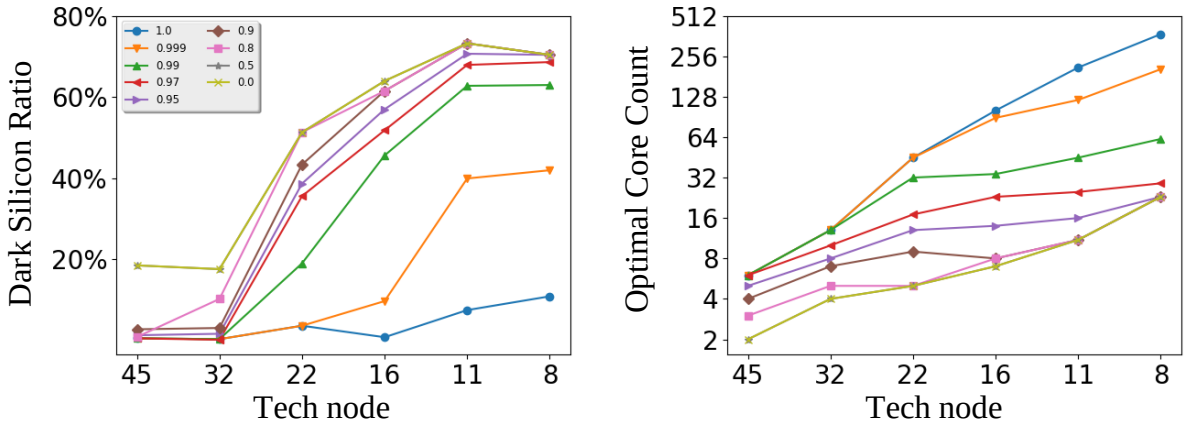


Figure 3.1: Upper-bound ITRS scaling with symmetric topology.

3.1.3 Unstructured High-level Architecture Modeling Pitfalls

Building and executing an architectural model with an unstructured approach (e.g., in a spreadsheet or some general purpose scripting language) is clearly possible[2], but the lack of a common abstraction introduces some issues as one tries to scale the analysis. Each additional interacting component is a set of new opportunities to make an uncaught mistake.

The degree to which these mistakes end up in the final model (and the amount of effort required to make sure it is mistake-free) is a function of the degree of composability, reusability, consistency and completeness checking supported by the tool. It is easiest to see this if we talk specifically again about the code of our example dark silicon analysis.

We first note that, although clearly defined conceptually, the three models needed are each of a different *form*: *DevM* is essentially a table of different scaling factors, *CorM* is an empirical set of points and a regression curve and *CmpM* is in the form of mathematical equations relating a set of high-level architectural quantities. Furthermore, even if they were of the same form, they are not "functions" but rather a set of mathematical *relationships*. The distinction is quite important. With traditional lvalue / rvalue style assignments (common to both functions

[2]With all the potential issues, unstructured methods in architectural modeling may not be as correct as one tends to believe [67, 68].

```
# Type definitions.
# A real number greater than 0.
typedef R+ : Real r
    r > 0

# A real number between [0, 1].
typedef Fraction : Real f
    0 < = f, f < = 1

# Simple Fit of the ITRS Scaling (DevM).
define ITRS:
    ref_tech_node : R+ as ref_t
    ref_core_performance : R+ as ref_perf
    ref_core_power : R+ as ref_power
    ref_core_area : R+ as ref_area
    tech_node : R+ as t
    core_performance : R+ as perf
    core_power : R+ as power
    core_area : R+ as area
    perf_scaling_factor : R+ as a
    power_scaling_factor : R+ as b
    ref_t = 45
    perf = a * ref_perf
    power = b * ref_power
    area / t**2 = ref_area / ref_t**2
    a = piecewise((1., t=45),   (1.09, t=32),
                  (2.38, t=22), (3.21, t=16),
                  (4.17, t=11), (3.85, t=8))
    b = piecewise((1., t=45),   (0.66, t=32),
                  (0.54, t=22), (0.38, t=16),
                  (0.25, t=11), (0.12, t=8))

# Pollock's Rule Extended with Power (CorM).
define ExtendedPollacksRule:
    ref_core_performance : R+ as perf
    ref_core_area : R+ as area
    ref_core_power : R+ as power
    area = 0.0152*perf**2 + 0.0265*perf + 7.4393
    power = 0.0002*perf**3 + 0.0009*perf**2
            + 0.3859*perf - 0.0301
    perf < 50

```

Listing 3.1: Dark silicon analysis on symmetric topology with ITRS scaling (continue in Listing 3.2).

```
# Amdahl's Law under Symmetric Multicore (CmpM_U).
define SymmetricAmdahl:
    speedup : R+ as sp
    core_performance : R+ as perf
    core_area : R+ as a
    core_power : R+ as power
    core_num : R+ as N
    chip_area : R+ as A
    thermal_design_power : R+ as TDP
    fraction_parallelism : Fraction as F
    dark_silicon_ratio : Fraction as R
    sp = 1 / ((1 - F) / perf + F / (perf * N))
    N = min(floor(A / a), floor(TDP / power))
    R * A = A - N * a

# Assumptions are now explicit and composable.
given ITRS, ExtendedPollacksRule, SymmetricAmdahl
assume chip_area = 111.0
assume thermal_design_power = 125.0
assume fraction_parallelism = [0.999, 0.99, 0.97,
                               0.95, 0.9, 0.8, 0.5]
assume tech_node = [45, 32, 22, 16, 11, 8]
assume ref_core_performance = linspace(0, 50, 0.05)
explore speedup, dark_silicon_ratio, core_num
```

Listing 3.2: Dark silicon analysis on symmetric topology with ITRS scaling (continued).

and spreadsheets) you end up with four issues:

Composition: It is hard to link the models' I/O together or even check if the models can be connected properly at all. Architectural models usually are connected to each other through some common system parameters or physical quantities. In this example, to do the dark silicon analysis, one needs to take scaling factors from tables in $DevM$, pass them as inputs to $CorM$, apply the values and re-fits the curve for different *tech node*, after which one then has to sample from the two Pareto curves in $CorM$ and supply the samples to $CmpM_U$ for final evaluation. This chain of data movement and dependency is not explicitly exposed by the models, and it takes some effort to understand how these models link together. This issue of mismatched form is even more acute when one wishes to switch out the $CmpM$ core model with the $CmpM_R$ core model because $CmpM_R$ takes a completely different set of inputs. With unstructured methods, one has to explicitly program these connections typically by function call chains. With Charm, one simply specifies all variables upfront within each model, and as long as the full variable names are consistent, Charm "wires up" the models through these channelling I/O variables. Perhaps most importantly, Charm throws an error when the models cannot be properly linked. For example, if one forgets to provide values for technology node (Line 64), Charm will complain that too many variables are free, or if the scaling model is about transistor rather than processor core, as long as the variables are properly named (e.g., one does not name transistor performance as *core* performance), Charm will capture this mismatch and warn that the models cannot be connected.

Restructuring and Reorientation: The models cannot be evaluated in a flexible way. Even though the model is a relationship between quantities, in spreadsheets or scripting languages one has to implement the evaluation as functions with fixed arguments. In this example, one typically codes up to evaluate the *speedup* from given value of *core performance*. If the control quantity is changed to another, say *core area*, one has to fix the code. An even worse, and

probably more interesting, case is when the control becomes the one under investigation, i.e., the input/output of the functions are reversed. In our example here, it happens when one wishes to explore the core count constraint given a target dark silicon ratio. There is no easy way for ad hoc methods to deal with this kind of flexibility but to completely reprogram. While in Charm, models *are* interpreted as a set of *mutually* dependent relationships without a fixed direction, and Charm runtime will generate the needed functions based on the provided controls and the quantities to explore.

Reasoning under Uncertainty: Architectural models usually involve some uncertainties [19], such as how technology may scale over the next 10-15 years. It is natural for computer architects to first evaluate the model with some concrete values (e.g., the scaling factors in Line 26, 29) and then model the uncertain quantity as some distribution, e.g., Gaussian distribution, as in our case studies in Section 3.5. It requires non-trivial programming effort with spreadsheets and scripting languages to support uncertain random variables. Charm supports different forms of input values such as scalars, vectors as well as distributions to ease architectural exploration.

Exploration: The analysis procedure is often coupled with the model definition. A common practice for computer architects is to explore the design space by iterating over a set of design options or different values for some system configuration knobs. With the high-level models, architects usually write imperative instructions to iterate over specific variables, and when the iterative variable changes to another, it quickly becomes tedious and error-prone to break and reconstruct the many-fold nested *for* loops. Charm decouples the model specification (Line 11-56) from the analysis procedure declaration (Line 59-66). Such iterations over input values are declarative and transparent (as opposed to writing *for* loops imperatively) by simply providing a list of values as inputs (Line 62, 64 and 65) in Charm.

Secondly, computer architectural quantities often have certain physical meanings. For example, *core performance* typically cannot be negative. A potential issue with unstructured

methods is that these boundaries are usually only programmed ad hoc in spreadsheets or scripting languages. A negative *core performance* may be totally *mathematically* valid and *will* lead to meaningless misleading result if not captured in the unstructured implementation. This issue is even more likely to occur in the following two cases.

Implicit Domain Constraints: Architectural models typically have their range of operation. Aside from the physical constraints, implicit domain constraints also come from how the model is built at first place. In the dark silicon example, the normalized performance of the real data points that the authors used to generate the $CorM$ is in the range of $(0, 50)$. Even though one can argue that a core with *normalized performance of 100* generally follows that regression but the result derived from that is much less accurate and trusted. This type of constraints are at most times only implicitly conveyed through the model building process, where it leads to a potential pitfall when the model is reused, especially when one only tries to interpret and re-implement the model from natural language descriptions (like in a published paper). While Charm encourages model builders to put in these implicit constraints explicitly as constraints built in the model specifications, e.g., Line 41. Charm will automatically check to see if these constraints are violated during evaluation.

Unbounded Distributions: Many architectural quantities follow normal distribution such as *core frequency* due to process variability [53–55]. When using these types of unbounded distributions, it sometimes violates the physical constraints of the quantity (*frequency* must be positive). In unstructured modeling, this check is completely ad hoc and, if overlooked, will lead to meaningless results. With Charm, this issue is automatically handled by the type checker, as long as one specifies a correct type for the quantity, e.g., *frequency :* $R+$.

Last but not least, the design space to cover is typically huge with high-level models. In the dark silicon model, the authors explore a hundred core configurations for each combination of a scaling trend in $DevM$ and a CMP model from $CmpM_U$ or a workload with $CmpM_R$. The

models are often to be evaluated hundreds of thousands, if not millions, of times which will take a non-trivial amount of time. It only becomes worse when one tries to evaluate models with uncertainties [19]. Without a structured system, a quick spreadsheet or naive prototyping will end up with unacceptable performance when the problem is scaled up and the burden of optimization falls upon the model builders and others who wish to use existing models through re-implementation. As we show in Section 3.5, with the invariant hoisting and memoization techniques, Charm greatly speeds up the exploration without additional effort from the model builders.

3.1.4 Benefits of Charm

Charm addresses all the above issues and serves as a unified layer for the representation, execution, and optimization of closed-form high-level architecture models. Charm provides a concise and natural abstraction to clearly express architectural relationships, automatically check model consistency and easily declare analysis goals. Charm can also transparently search the design space for optimal configurations utilizing state-of-the-art constraint solvers. Building and evaluating closed-form high-level architecture models using Charm has the following major benefits:

Clarity through abstraction – Charm encapsulates a set of mutually dependent relationships and supports flexible function generation. It enables representation of architecture models in a mathematically consistent way and modulates high-level architecture models by packing commonly used equations, constraints and assumptions in modules. These architectural "rules of thumb" can then be easily composed, reused, and shared in a variety of modelling scenarios.

Flexibility through automation – Rather than treating the mathematical relations as functions, like in traditional programming languages, Charm keeps the abstraction at the mathematical level; hence, it is able to generate corresponding dataflow graph on-the-fly without requiring

the user to re-write the model when the same model is used for different high level studies. To further assist automatic design space exploration, we extend core Charm to transparently transform the system model into an SMT instance, if it is under-determined (there are one or more free variables in the model), and utilizes SMT solvers (i.e., z3 [69] in our implementation) to efficiently explore the design space through bounding the infinite search space and approximation.

Safety through type-checking – Charm enables new static and run-time checking capabilities on high-level architecture models by enforcing a type system. One example is that many architecturally meaningful variables have inherent physical bounds that they must satisfy; otherwise, although mathematically viable, the solution is not realistic. With the type system built-in, Charm can dynamically check if all variables are within user-defined bounds to ensure a meaningful modelling result. The type system also helps prune the design space based on the bounds, without which a declarative analysis might end up wasting a huge amount of computing effort in less meaningful sub-spaces. Charm also incorporate physical unit as an optional part of variable definition and will check and convert physical units dynamically.

Efficiency through optimization – Charm opens up new opportunities for compiler-level optimization when evaluating architecture models. Although high-level architecture models are usually several orders of magnitude faster than detailed simulations, as the model gets complicated or is applied many times to estimate a distribution, it can still take a non-trivial amount of time to naively evaluate the set of equations in every iteration. By expressing these complicated models in Charm, we are able to identify common intermediate results to hoist outside of the main design option iteration and/or apply memoization on functions.

Finally, and perhaps most importantly to the community, Charm promotes collaboration between application designers, computer architects, and hardware engineers because they can now share and refine models using the same formal specification and a common set of abstrac-

tions. For example, to reason about the energy consumption of an application on a platform, with his/her own application model written in Charm, the application developer won't have to implement an energy model from scratch and can simply plug in an existing one written also in Charm.

3.2 Related Work

There exist systems and languages that support structured analytical modeling. Modelica [70] supports multi-domain analytical modeling with an emphasis on object-oriented model composition, but the connection of models need to be explicitly dictated and the design space exploration requires user intervention. On the other hand, Charm is more restricted, and thus it is able to automatically link models and generate exploration loops. Aspen [71] provides a DSL to express applications along with an abstract machine organization in order to model performance. Palm [72] utilizes source code annotation to build analytical models for target applications. LSE [73] is a fully concurrent-structural modeling framework designed to maximize reusability of components. There are also many other works in the field of HPC for automatically performing performance modelling [74–76]. Most of these languages and systems serve a different purpose of expressing the mapping between performance/power models and specific detailed application/architecture and are not well-suited for high-level analytical design space exploration. In contrast, Charm is tailored for structured yet flexible exploration of the interactions between architectural variables as well as their ramifications at a high level. There are also a few systems exploiting the power of symbolic execution for modeling [19,67], but Charm provides more capabilities around formalizing, checking, and evaluating the models. There also exists a tool [77] of the same name CHARM (Chip-architecture Planning Tool) that uses a knowledge-based scheme to ease high-level synthesis.

The internals of Charm resemble some of the data-flow centered programming languages

$$
\begin{aligned}
var, rn, tn \in Name \qquad & rel \in Relation \\
val \in Value & \\
p \in Program & := \overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var} \\
td \in TypeDefinition & := \textbf{typedef}\ tn\ \overrightarrow{rel} \\
rdef \in RuleDefinition & := \textbf{define}\ rn\ \overrightarrow{rdecl} \\
rdecl \in RuleDeclaration & := var\ tn \mid rel \\
a \in AnalyzeStatement & := \textbf{given}\ \overrightarrow{rn} \mid \textbf{assume}\ \overrightarrow{asmt} \\
asmt \in Assignment & := var = val
\end{aligned}
$$

Figure 3.2: Abstract syntax of charm. A program is a sequence of type definitions, rule definitions, analysis statements, and a list of variables to explore. Relations are atomic with respect to the semantics; they use the syntax and semantics of the backend solver. They use the standard arithmetic and comparison operators, and allow lists, tuples, and real numbers as possible values.

in the field of incremental/reactive programming [78–82] but differ in that Charm is highly restrictive. This restrictiveness means that Charm is more of a modeling language rather than a programming language, i.e., Charm does not support general purpose structures like loops and function calls but supports a malleability useful for exploration (e.g., reversing input/output dependencies).

3.3 Supporting Analytical Modeling from A PL Perspective

Charm provides a simple domain specific modeling language to express both closed-form models and the design space exploration dimensions. The DSL has an easy-to-use Python-like syntax. In terms of mathematical expressiveness, Charm supports all common closed-form algebra that computer architects often resort to, including linear algebra like polynomials and simple non-linear algebra like exponentiation. Basic non-closed-form functions like summation and product are also supported. To enhance the design space exploration to uncertain do-

$$C, D, E, \Omega \in RelationSet \quad \Gamma \in TypeEnvironment = Name \to RelationSet$$
$$\Theta \in RuleEnvironment = Name \to RelationSet \quad \mu \in VariableMap = Name \to Value$$

$$\frac{C = \left\{c \mid c \in \overrightarrow{rel}\right\}}{\textbf{typedef}\ tn\ \overrightarrow{rel} \Downarrow_T (tn, C)}\ \textsc{typedef} \qquad \frac{(\Gamma, rdecl_i) \Downarrow C_i \quad C = \bigcup C_i \quad i \in 1..|\overrightarrow{rdecl}|}{\left(\Gamma, \textbf{define}\ rn\ \overrightarrow{rdecl}\right) \Downarrow_R (rn, C)}\ \textsc{ruledef} \qquad \frac{\Gamma(tn) = C}{(\Gamma, var\ tn) \Downarrow C\,[var/tn]}\ \textsc{rd-var}$$

$$\frac{}{(\_, rel) \Downarrow \{rel\}}\ \textsc{rd-rel} \qquad \frac{C_i = \Theta(rn_i) \quad C = \bigcup C_i \quad i \in 0..|\overrightarrow{rn}|}{\left(\Theta, \textbf{given}\ \overrightarrow{rn}\right) \Downarrow_A C}\ \textsc{given} \qquad \frac{}{\left(\_, \textbf{assume}\ \overrightarrow{asmt}\right) \Downarrow_A \left\{e \mid e \in \overrightarrow{asmt}\right\}}\ \textsc{assume}$$

$$\frac{\begin{array}{c}\texttt{Ext}(x) = \emptyset \bigvee \texttt{Ext}(y) = \emptyset \bigvee \texttt{Ext}(x) = \texttt{Ext}(y), \forall x, y \in \texttt{vars}(rel) \\ \omega = \left\{\alpha(a) \mid a \in \bigcup \texttt{Ext}(b_i), \forall b_i \in \texttt{vars}(rel)\right\} \quad \alpha(a) = rel[x.a/x, \forall x \in \texttt{vars}(rel)]\end{array}}{rel \Downarrow_M \omega}\ \textsc{multi-instance}$$

$$\frac{\begin{array}{ccc}\Gamma(tn_i) = C_i \text{ where } td_i \Downarrow_T (tn_i, C_i) & & \Theta(rn_j) = D_j \text{ where } \left(\Gamma, rdef_j\right) \Downarrow_R (rn_j, D_j) \\ \Omega = \bigcup E_k \text{ where } (\Theta, a_k) \Downarrow_A E_k & \Omega' = \bigcup\{\omega \mid rel \Downarrow_M \omega \bigwedge rel \in \Omega\} & \texttt{isConsistent}(\Omega') \\ \texttt{isFullyDetermined}\left(\Omega', \overrightarrow{var}\right) & \mu = \texttt{SOLVE}\left(\Omega', \overrightarrow{var}\right) & i \in 1..|\overrightarrow{td}| \quad j \in 1..|\overrightarrow{rdef}| \quad k \in 1..|\overrightarrow{a}|\end{array}}{\overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var} \Downarrow_P \mu}\ \textsc{program}$$

Figure 3.3: Operational semantics of Charm. Relations are here taken as atoms; they use the semantics of the backend solver engine. An overhead arrow indicates a sequence of one or more elements. $C[x/y]$ indicates to substitute all instances of y in C with x. `vars` returns the names of all variables used in the relation set, while `Ext` returns all extensions of a variable (portion of the name appearing after a dot when multi-instanced). `isConsistent` ensures the relation set is consistent. `isFullyDetermined` ensures the relation set is fully determined with respect to $\overrightarrow{var}$. `SOLVE` is an instance of the backend solver; it returns a mapping of all specified variables to values (real numbers, lists, and tuples). TYPEDEF takes a type definition and returns a tuple with type name and relation set. RULEDEF takes a rule definition and the type environment and returns a tuple with rule name and relation set. RD-VAR takes a type rule declaration and the type environment and returns a relation set, where relations on the indicated type now apply to the indicated variable. RD-REL takes a relation rule declaration and returns the same relation in a set. GIVEN takes a **given** analyze statement and the rule definitions and returns the relation set of the indicated rule. ASSUME takes an **assume** analyze statement and returns a relation set of all the declared equalities. MULTI-INSTANCE takes a relation and returns a set of relations, where the original relation is duplicated once for each extension possessed by its variables, with the names of the variables replaced by their extended version (as discussed in section 3.3.2). PROGRAM takes a program and returns a map for the list of exploration variables, mapping each to real numbers, lists, and tuples determined by the backend solver.

mains, Charm also supports distributional values to be set and propagated through the models transparently. Once written in Charm DSL, the interpreter is able to transform the mathematical relationships and constraints into a series of data-flow graphs for fast evaluation. A type system is applied to make sure all architecturally meaningful quantities operate in the correct domain. Charm also optimizes the design space exploration procedure using compiler techniques to eliminate redundant computation. Figure 3.4 graphically shows the interpretation process.

In this section, we first describe the abstractions Charm provides and formalize the syntax and semantics of Charm DSL. We then articulate the internal design of the interpreter and how type checking, definability checking, evaluation and optimization are done in Charm.

3.3.1 Language Abstractions

Charm provides a common layer with three key abstractions to address all the potential issues in Section 3.1.3. In Charm DSL, five keywords are reserved to express three abstractions: **types**, **models** and **analysis**.

Keyword *typedef* translates into the first abstraction: **type**. The type system is designed to be simple but useful: each type is essentially a base type with constraints, e.g., *R+* is defined as a positive number of base type *real* in Listing 3.1 Line 3-4. There are only two base types, *Real* and *Integer* standing for real numbers and integer numbers respectively. Internally, real numbers are represented by *float* and integers by *int*.

The second key abstraction is **model**. Keyword *define* constructs a model. A model specification in Charm encapsulates the following two pieces of information in a high-level architecture model:

- **A set of variables.** Each variable has a universally unique and consistent full name. Each variable also has a local short name (optional), as well as an explicitly declared type and

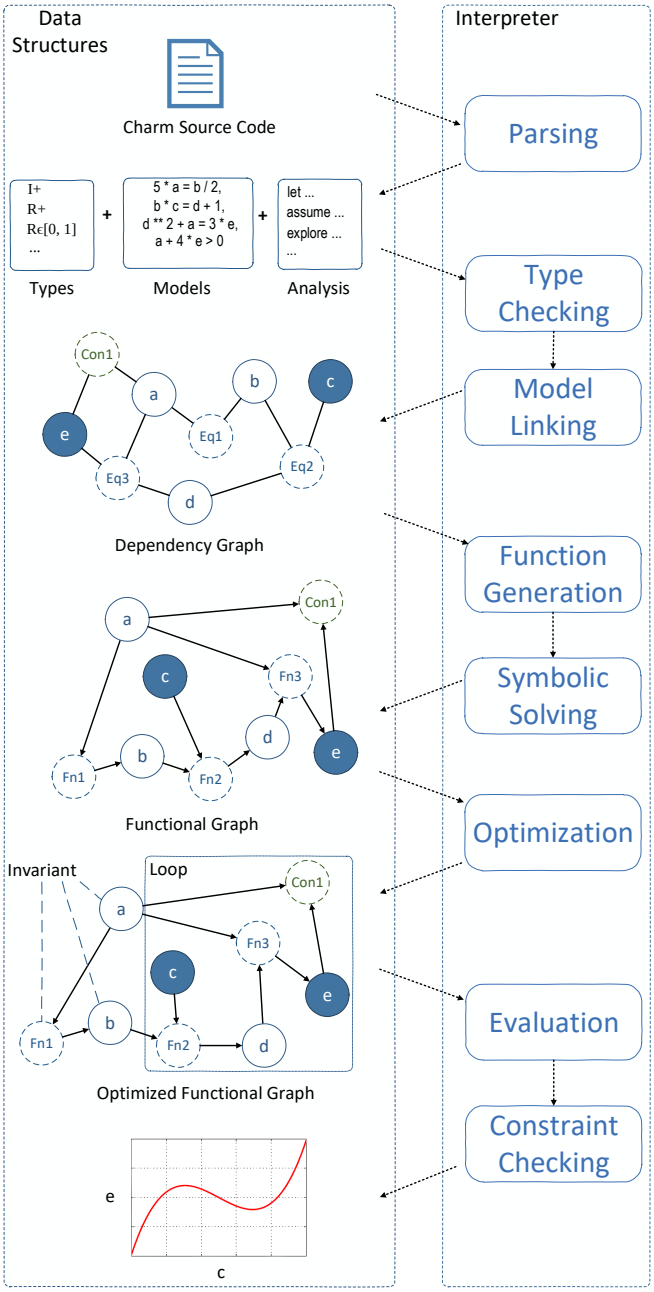


Figure 3.4: Overview of Charm interpreter. The example system has 3 equations (Eq1, Eq2 and Eq3), 1 constraint (Con1) and 5 variables in which c is the iterative input. In this example, we are tying to explore the relationship between e and c given a. The parser takes Charm source code and breaks it into a set of types, a set of model definitions and a set of analysis statements. Charm then links types, models and assignments in a dependency graph after they go through the type checker. The graph then is fed to a function generator and a symbolic solver to convert it into a functional graph. The optimizer finally takes the functional graph and annotate it with hints for execution before it finally gets evaluated and checked against model constraints.

physical unit (optional, e.g., chip_area : R+ in μm^2). For instance, in Listing 3.1 line 35, the short name is "*perf*", the type is "R+", which means positive real number, and the long name is "*ref_core_performance*". Short names live only within the definition of a model, while full names are exported to other models as well as to the analysis logic.

- **A set of relations.** In Charm, both equations and inequalities are considered *relations* . Relations define mathematical relationships between variables using either their full or short names (e.g., Listing 3.1 Lines 41, 54-56). Both linear and nonlinear systems are present in architectural models that we care about. The general problem of determining the definability of and solving such systems is theoretically hard and beyond the scope of this work. Given the limitations and solving capabilities of existing back-end solvers, some very complicated non-linear equations cannot be symbolically solved (e.g., Charm will throw an error if one tries to solve for x in $y = (a^{1/x})^{2^x}$). Fortunately though, we find that most models computer architects care about are well within the limit. Equations can also bind variables to constant quantities as assumptions defined in the model specification (e.g., $kBoltzmann = 8.6173303 \times 10^{-5}$). Relations can be value generative, if values of all but one variable are given. A relation can also be non-value-generative, which includes all inequalities and equations that have all their variables assigned a value.

Charm DSL accepts different mathematically equivalent forms of relations, so that different modelers with different background expertise can write the math in the conventional way of their own fields and use other models directly as they are without rewriting.

The Charm DSL is strongly typed. The model abstraction enforces explicit type declaration to make sure there are not implicit assumptions about data types and domains across models.

Charm abstracts the common structure of an **analysis** with three keywords: *given*, *assume* and *explore*.

Before computation, *given* statement selects the model in the analysis. If multiple models

are selected, they are linked together automatically by the interpreter. Full names of variables are used to connect each other across models.

Although in general, many algebra systems can be solved without additional inputs, for computer architecture models, at most times, some control quantities need to be given (e.g., design options like *core size* and system configurations like *cache associativity*) in order to solve for the quantities under investigation (e.g., *speedup* of a CMP). Keyword *assume* serves such purpose by differentiating assignment equal signs from mathematical equal signs inside model specification, i.e., *assume* statements are assignments much like in other programming languages while equations in model specification are merely mathematical relationships which do not imply a direction of data movement. Charm also constrains *assume* statements to be assignment with constants, i.e., they can only be used to express external inputs to the model rather than defining additional relations outside of the model specification.

Charm supports both scalar and vector value assignments, as well as random variable of commonly used distributions, e.g., *Gaussian Distribution*.

Iteration is expressed in a Pythonic list-like syntax or functions that generates a list, e.g., *linspace*, and assigned to some input variable just like a normal *assume* statement (e.g., Listing 3.1 Line 65). Charm handles iteration naturally by selecting combinations of all iterative input values non-repeatedly from their Cartesian space in a Gray code fashion. Two special cases are: a), if two or more input variables are dependent, they can be expressed like Python tuple assignment, e.g., $assume$ ($tech\_node$, $freq\_scaling\_factor$) = [(45, 1.), (32, 1.09)] and b), if a variable is indexed, it can be expressed using special "list" notation after its variable name, e.g., $assume\ L[] = [1, 2]$, which means $L[0] = 1$ and $L[1] = 2$. These notations become handy when we write the quantum models with Charm in Section 3.5.2.

Finally, an analysis is completed by specifying which quantities to solve for symbolically and evaluate using *explore*. Charm exploits a data-flow centric approach and builds a directed acyclic functional graph internally to propagate given values through linked models to the final

responsive variables architects wish to explore.

Figure 3.2 gives the abstract syntax of Charm and Figure 3.3 formalizes the semantics.

3.3.2 Language Internals

In order to evaluate the models and optimize the evaluation logic, Charm uses two data-flow graph structures internally to represent and transform the computation. In this section, we first define the core graph data structures and then describe how we can perform type checking, function generation, evaluation and optimization with these graph structure.

Dependency Graph. A dependency graph is a bipartite graph $G =< V_{var}, V_{rel}, E >$, where:

- V_{var} is the variable node set in which every variable in the selected models is a vertex.
- V_{rel} is the relation node set and $V_{rel} = V_{eq} \cup V_{con}$, where V_{eq} is the set of vertices in which every equation in the selected models is a vertex; V_{con} is the set of vertices in which every constraint in the selected models is a vertex.
- E is the set of edges and there exists an edge between vertices in V_{var} and V_{rel} if and only if the variable name appears in the relation.

Functional Graph. A functional graph is a *directed acyclic* dependency graph D in which:

- Every node in V_{var} has at most 1 incoming edge, i.e., its in-degree being 0 or 1.
- Every node in V_{eq} has at most 1 outgoing edge, i.e., its out-degree being 0 or 1.
- Every node in V_{con} has no outgoing edge, i.e., its out-degree being 0.

Dependency graph building and static type checking. To build the dependency graph from the models, Charm performs a single scan over all relations in the models. It assigns a variable node to every variables with a unique full name (including variables automatically generated by

multi-instancing) and an equation/constraint node to every equation/constraint. When creating relation node, Charm creates an edge between the equation/constraint node to a variable node if the variable is used in the equation/constraint. Finally, Charm scans the analysis statements and marks variable nodes being assigned as input nodes.

Charm performs simple type checking both statically when building the dependency graph after parsing and dynamically when checking constraints at runtime. Static type checking is done by tracking the variable names and types when building the dependency graph. Each variable must be declared with an explicitly defined type. If a variable name is used by two or more relations, we check that their defined types are identical (both base type and constraints associated). Charm aborts execution and issues an error message for inconsistent types.

Relation Multi-instancing. When building a dependency graph, different variables sometimes follow the same mathematical relationships. An example is *core_performance.big* and *core_performance.small* defined in Listing 3.3 Line 5-6. Both of them follow equation in Listing 3.1 Line 23 when plugged in for evaluation. We discuss their physical meanings later in Section 3.5.1, but they are essentially two variables following the same mathematical relationship. We refer this behavior as "relation multi-instancing" and use the dot notation (a variable name and a name extension concatenated by dot, e.g., *core_area.big*) to invoke multi-instancing. Charm internally creates variable nodes and relation nodes for multiple instances with different name extensions. Figure 3.5 shows how these nodes in the dependency graph are created. The model is ill-defined if Charm fails to find extended input variables with consistent name extensions or discovers inconsistent name extension sets for different variables trying to invoke multi-instancing.

Functional graph building and function generation. After building the dependency graph G, the function converter tries to convert G into a functional graph F. If it can convert successfully, there is a viable solution when all equations or sets of equations can be solved and lambdified

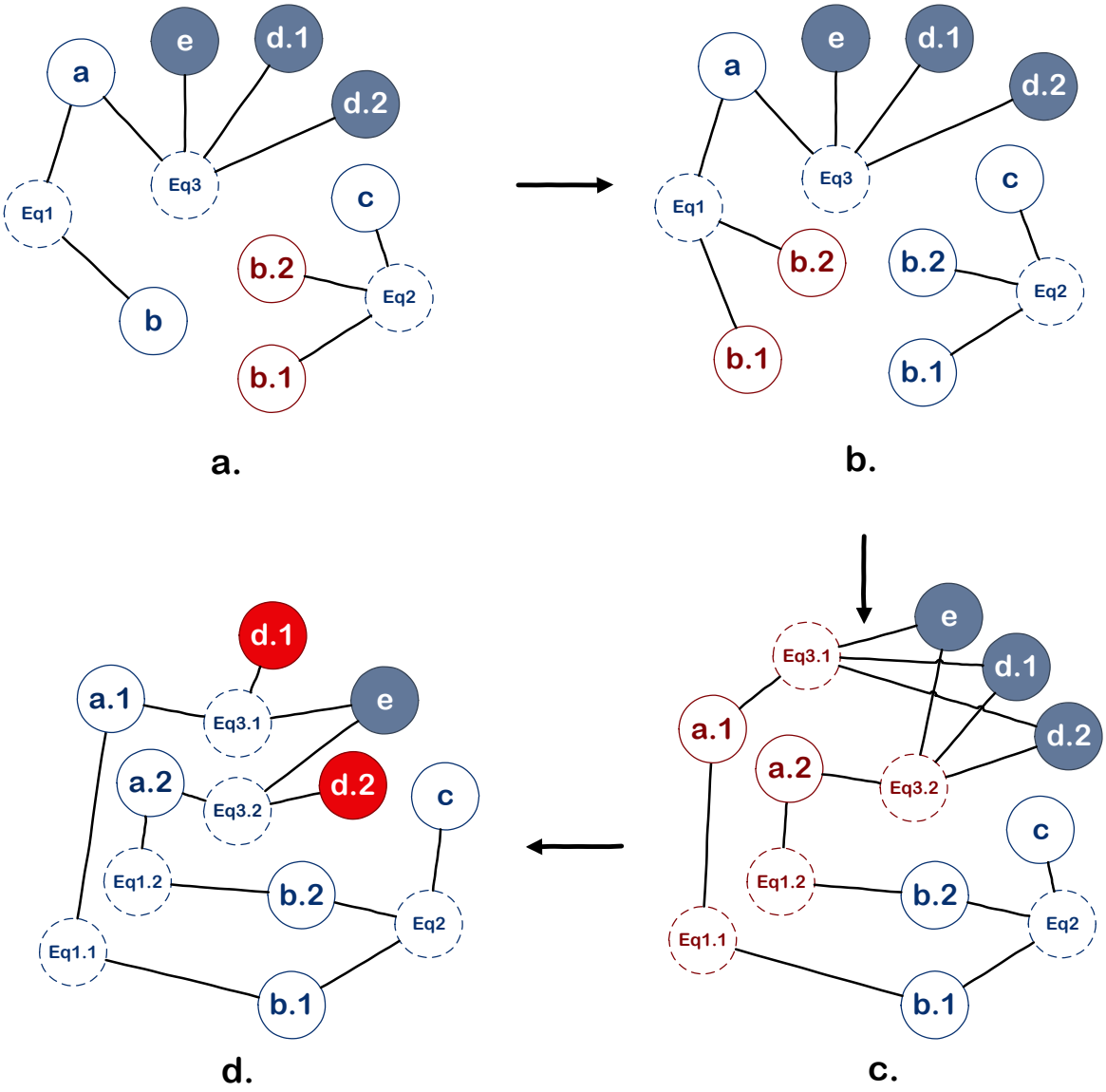


Figure 3.5: Relation multi-instancing when generating dependency graph. a) The initial graph has extended names ($b.1$, $b.2$). b) Charm finds and splits the corresponding base name node. c) Charm propagates the multi-instancing, i.e., all nodes connected to the base name node (b) are also split. Then Charm merges names with same extension together. d) The multi-instancing ends with checking input nodes for identical name extensions and removing edges between non-consistent name extensions. In this case, it ends when the split process reaches d and e, successfully finds $d.1$ and $d.2$ which are extended names with consistent name extension set ($\{.1, .2\}$ in this example) and removes the edges between ($d.1$, $Eq3.2$) and ($d.2$, $Eq3.1$).

by the back-end symbolic solver, and therefore the models can be evaluated by Charm.

The function converter backtracks through G in a DFS manner and tries to label all the edges with a direction without introducing a conflict. A conflict occurs when an equation node has more than one outgoing edges or when an inequality node has any outgoing edge or when a variable node (excluding input nodes) does not have exact one incoming edge. If there is a successful labeling of all edges, Charm uses Sympy [59] as the back-end solver to convert all equations and constraints (all inequalities and equation nodes with an out-degree of 0 are considered as constraints at this point) into callable functions with inputs being the variables directly pointing to the equation/constraint and output being the variable pointed at by the equation node. As part of type checking, each variable node is also associated with the constraint from its type. These type constraints are also lambdified and evaluated during evaluation. The search space for conversion is in practice greatly reduced by the following heuristics:

- All edges with one node being input node have fixed direction (from the input node).
- All edges with one node being a dangling variable node (variable node that has only one edge) have fixed direction (to the variable node).
- All edges with one node being a constraint have fixed direction (to the constraint node).

Cycle elimination. A functional graph F must be *acyclic* in order to evaluate. However, when there are codependent equations, they form cycles. In case of a cycle, all equation nodes in the cycle must be solved altogether. We pass the equations in a cycle to the solver at once and then replace the cycle with pairs of function node and variable node, where each pair is a mapping between all inputs to the cycle (a dummy input node is created if there are no inputs from other parts of the graph to the cycle) and one variable node inside the cycle. Each function node generated by the cycle has one variable along the cycle as its output and all functions

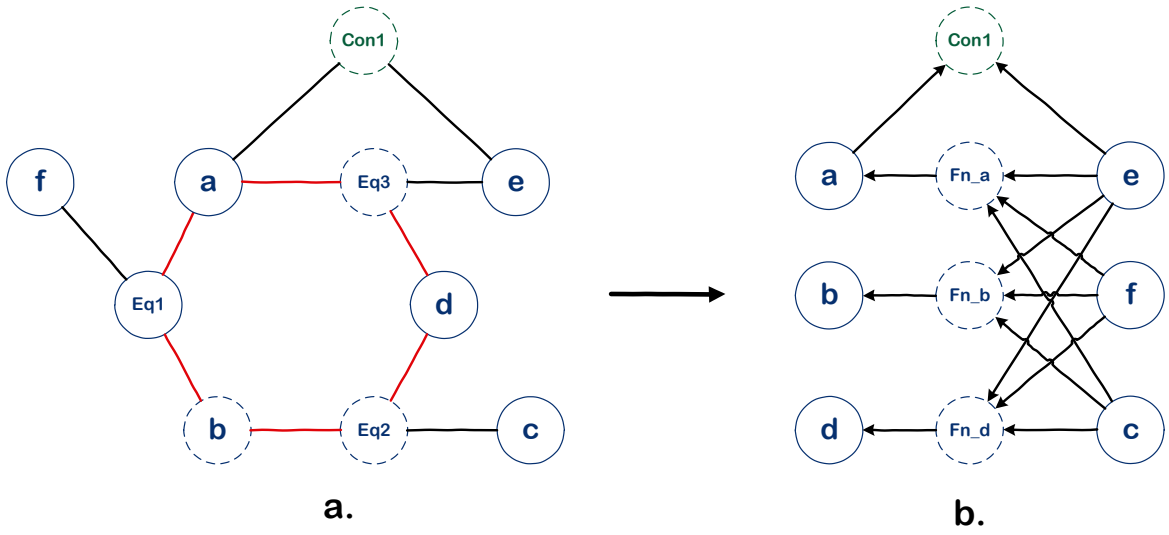


Figure 3.6: Cycle elimination when generating functional graph. Equations in a cycle are solved at once and are replaced with three functions, each of which generates a different variable value.

generated by the cycle are from the same set of equations, only with different variables as output. Figure 3.6 shows an example of cycle elimination in F.

Computational constraints. A special computational constraint is applied when building a functional graph: some mathematical operators are not reversible or have infinite solutions, such as $\sum$ and $\prod$, some are computationally hard for the solver, like solving x in $y = (a^{1/x})^{2^x}$. For the non-reversible equation, its direction is fixed, i.e. its edges have fixed direction not subject to the function converter.

Evaluation and constraint checking. Once we have a viable functional graph F, a feasible solution is to derive from all input nodes and propagate the given values by traversing F. Each following function/constraint node is transformed using higher-order functions to "remember" propagated partial values before all inputs are ready and it can be evaluated.

Optimization. Oftentimes architects explore the relationship between two variables by iterating over different input values. One simple yet effective optimization is invariant hoisting.

With the functional graph structure, it is straightforward to optimize for invariant in Charm. From each iterative variable node, Charm simply traverse from that node, then all nodes that cannot be reached from the iterative input nodes are invariant to iteration over that input. In the simple illustrative example in Figure 3.4, *c* is iterative and *a*, *b*, *Fn1* are invariant because there are not paths from *c* to them.

Each function node also caches a mapping table between inputs and its output. Such memoization optimizes away unnecessary re-computation over same set of input values.

3.4 Extensions to Core Charm

The core Charm we described in Section 3.3.2 provides a systematic way of managing closed-form architecture models. Through the abstractions of core Charm, we already achieved the many benefits including clarity, flexibility, safety and efficiency that we desired in Section 3.1.4. However, the core language solves the model (i.e., derives quantities queried) only when it is mathematically determined and solvable.

As the design shifts towards a landscape of less understood ASICs and beyond Moore technologies, unlike traditional systems such as CPUs where architects have tens of years of existing designs and experience to draw upon, we usually are faced with a large set of free variables both at the architecture level (e.g., issue width, buffer sizes) and application level (e.g., neural-network structure, tiling configuration), as well as a broad set of constraints (e.g., thermal limits, classification accuracy). The sheer number from the combinatorics of these parameters imposes a significant computational challenge when searching for an optimal design. Sweeping the vast space, even in closed-form only, incurs high costs in both time and resources.

To demonstrate the extensibility of Charm, we extend integrate Charm with z3 [69] to transparently support such effort by transforming an under-specified model into an SMT instance.

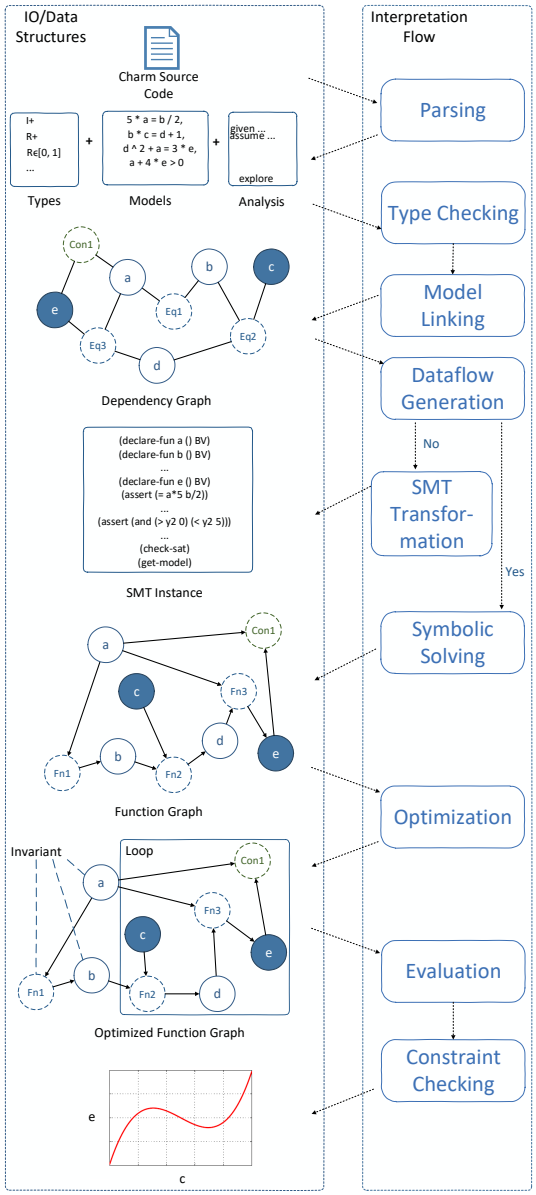


Figure 3.7: Overview of extended Charm. Difference between the extended workflow here and Figure 3.4 is the following: if the conversion from dependency graph to function graph is unsuccessful, Charm transparently generates an SMT instance based on the variables and relations in the model specification and calls out to the SMT solver to find a possible solution.

Figure 3.7 demonstrates this extension in the core Charm workflow. With any (or all) of the assumptions (Lines 60-65) taken away, Charm now automatically detects the free variables, transforms the relationships into bounded SMT instances, and calls out to z3 to search for a viable configuration (if there exists a satisfiable solution). Furthermore, Charm can also iteratively optimize the design by tightening constraints in the SMT instance (e.g., lower bound on performance, upper bound on power) on-the-fly.

Transforming under-determined system into SMT problem. If the conversion from dependency graph to function graph fails, i.e., the algorithm does not find a matching with a size equal to the number of variables in R, Charm aggregates all relations and exports a quantifier-free SMT instance to z3. It is well-known that some of the SMT theories are undecidable but, fortunately, the theory of bit-vectors and floating-points is bounded and decidable. We approximate each real variable in the model into a floating-point (fp) using the IEEE 754 encoding since most of time we do not need infinite precision (e.g., we typically do not need an IPC to the 10th decimal), and each integer variable is approximated by a bit-vector (bv). The transformation also bounds the search space by the number of bits we use. We set 32 bits as default length because it achieves good balance between applicable range (most design configurations fit within the range) and synthesis speed. The computation is done in the fp domain and we dynamically cast bv to/from fp values by rounding to the nearest even. We find in our experiments that this approximation works well for a wide variety of architecture models.

Optimizing under-determined systems. In a complex system design space, sometimes a manual search (even from a synthesized configuration) is not favorable as it can quickly become tedious and inefficient if each iteration requires a combination of hundreds of parameters. With the SMT solver, the optimization can be automatized by iteratively tightening/loosing bounds of the system constraints, e.g., iteratively asking the question "is there a configuration that has better performance?" or "is there a configuration that consumes less power/energy?". To

quantify how much better each iteration should be targeting, Charm allows users to specify a step coefficient (e.g., speedup@0.1) to control the granularity of the search.

3.4.1 Existing Design Space Support with Constraint Solving

There exists a line of research which utilize constraint-solving to guide design space exploration. Mohanty [83] uses constraint solving as a first step to prune the entire design space of embedded SoCs at a coarse granularity for later evaluation. CoBaSA [84] compiles the component-based software development design space and system constraints into a pseudo-Boolean satisfiability problem to find feasible solutions with a large number of constraints. Haubelt et. al. [85] encode system synthesis problem into a SAT instance in order to find a feasible binding between processes and resources. Regarding the use of SMT solvers [86–89], a plethora of research work has explored task scheduling and resource management using an SMT solver including methods with high-level language or custom DSL as the frontend [86, 89].

With Charm, we use SMT solver to explore the design space formed by analytic architecture models. We utilize the theories of bit-vectors and floating-point numbers to bound the infinite design space and approximate the architectural quantities. These SMT theories are seamlessly integrated with core Charm and provides a systematic way to express such underdetermined exploration queries.

3.5 Case Studies with Charm

In this section, we demonstrate the application of both core Charm and its SMT extension using three case studies. In the first case study, we demonstrate the benefits of Charm by extending the dark silicon analysis with a different topology and a distribution of technology scaling. We also compare the execution times with and without optimization.

The second case study focuses more on the problem of modeling a critical resource in fault-tolerant quantum computing and performs exploration with varying physical error rate. Interestingly, when validating Charm results in the second case study, Charm helps find inconsistent model definition errors, which are silently propagated through by Mathematica [90] and would have led to incorrect results.

The third case study demonstrates how extended Charm deals with under-specified system models and assists design space exploration with the use of SMT solvers. We use a well-cited analytic model for Convolutional Neural Networks (CNNs) [91] along with the roofline models of a set of FPGA platforms to explore tiling configurations for different CNN architectures.

3.5.1 Dark Silicon and Beyond

Listing 3.3 highlights all the changes that we need to implement in Charm to model and switch the DSE from symmetric topology to asymmetric. Note that in the asymmetric model, "relation multi-instancing" comes in handy when expressing two co-existing types of core. To switch the analysis, all we need to do is to change the models that are *given* (Listing 3.3 Line 22) and provide values for two types of core instead of one (Listing 3.3 Line 23-24). We also write a new constraint (Listing 3.3 Line 20) to specify the fact that the big core should have better performance than the small core.

It's even simpler to switch from ITRS scaling predictions to the conservative predictions [66]. Listing 3.4 shows all the changes needed. Figure 3.8 plots the resulting scaling trends for the asymmetric topology.

One interesting question one may ask is *"what if the actual technology scaling is somewhere in between the two predictions?"* We explore the design space with a distribution of scaling factors. We use a Gaussian distribution for the scaling factor, the mean of which being the average value between the two extremities and the standard deviation being the difference

```
# Amdahl's Law under Asymmetric Multicore (CmpM_U).
define AsymmetricAmdahl:
    speedup : R+ as sp
    # here we need two types of perf, area, power
    core_performance.big : R+ as big_perf
    core_performance.small : R+ as small_perf
    core_area.big : R+ as big_a
    core_area.small : R+ as small_a
    core_power.big : R+ as big_power
    core_power.small : R+ as small_power
    core_num : R+ as N
    chip_area : R+ as A
    thermal_design_power : R+ as TDP
    fraction_parallelism : Fraction as F
    dark_silicon_ratio : Fraction as R
    sp = 1 / ((1-F)/big_perf + F/(N*small_perf+big_perf))
    N = min(floor((A - big_a)/small_a),
            floor((TDP - big_power)/small_power))
    R * A = A - (N * small_a + big_a)
    big_perf >= small_perf

given ITRS, ExtendedPollacksRule, AsymmetricAmdahl
assume ref_core_performance.big=linspace(0,50,0.05)
assume ref_core_performance.small=linspace(0,50,0.05)
```

Listing 3.3: Asymmetric model and the changes in code.

```
# Conservative scaling model (DevM).
define ConservativeScaling:
    ...
    a = piecewise((1., t=45),   (1.10, t=32),
                  (1.19, t=22), (1.25, t=16),
                  (1.30, t=11), (1.34, t=8))
    b = piecewise((1., t=45),   (0.71, t=32),
                  (0.52, t=22), (0.39, t=16),
                  (0.29, t=11), (0.22, t=8))

given ConservativeScaling, ExtendedPollacksRule, AsymmetricAmdahl
```

Listing 3.4: Conservative scaling and the changes in code.

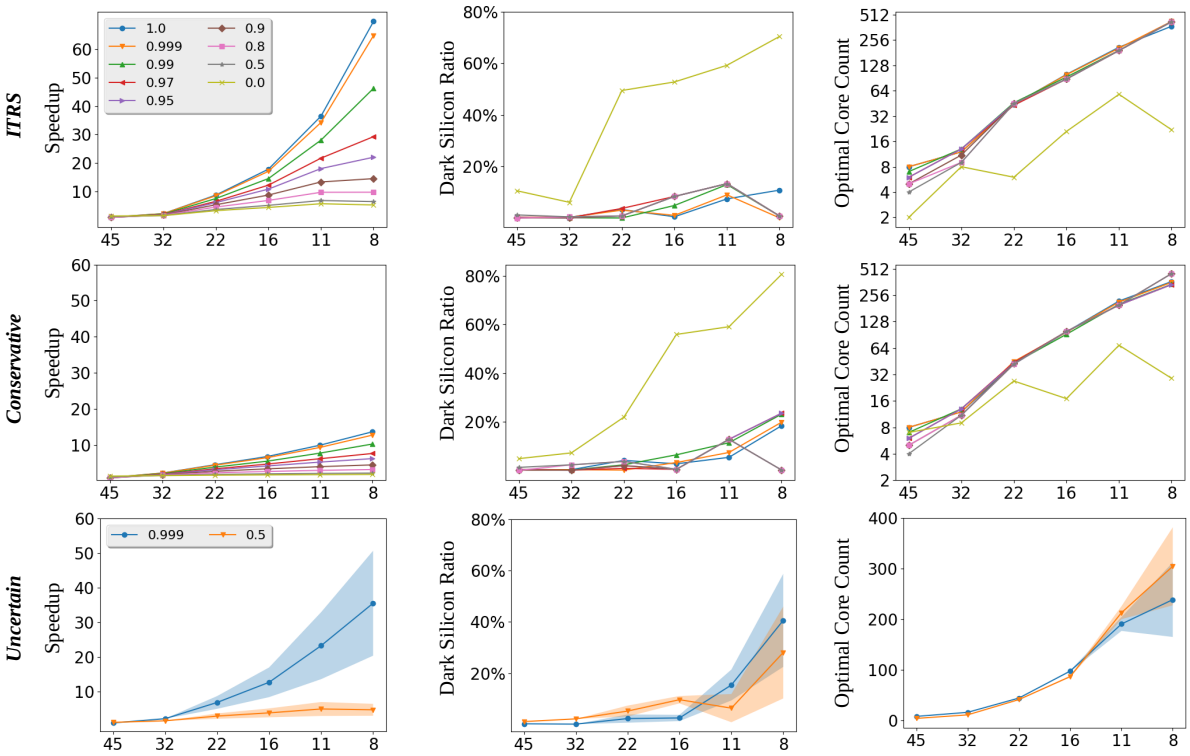


Figure 3.8: Upper-bound scaling with asymmetric topology with tech node on x-axis. Note that the last figure of optimal core count has a linear-scale y-axis to better demonstrate the variance. For clarity we only plot two regions in the uncertain scaling results, but the trends for other f values are similar.

```
# Distributional scaling model (DevM).
define DistScaling:
    ...
  a = piecewise((1.,t=45),(Gauss(1.095,0.005),t=32),
  (Gauss(1.785,0.595),t=22),(Gauss(2.23,0.98),t=16),
  (Gauss(2.735,1.435),t=11),(Gauss(2.595,1.255),t=8))
  b = piecewise((1.,t=45),(Gauss(0.685,0.025),t=32),
  (Gauss(0.53,0.01),t=22),(Gauss(0.385,0.005),t=16),
  (Gauss(0.27,0.02),t=11),(Gauss(0.17,0.05),t=8))

given DistScaling, ExtendedPollacksRule, AsymmetricAmdahl
```

Listing 3.5: Uncertain scaling and the changes in code.

between the mean and the extremities. Listing 3.5 shows the necessary changes in Charm code. It is important that although Gaussian distribution is not bounded, the scaling factors have a bounded domain. The type checking in Charm makes sure that the scaling factors a and b operate only in their defined domains (see Listing 3.1 Line 20-21), and the provided Gaussian distribution is converted to a truncated Gaussian distribution with the same mean and standard deviation within Charm. From Figure 3.8, we can see that with the technology scaling, the more parallel workload (with an f close to 1) shows more sensitivity towards technology uncertainties while the more serial workload is less sensitive to the changes in the core performance and power. Another probably even more interesting observation is that the optimal core count of the most performant configuration becomes very uncertain once we hit 11nm and beyond. The uncertainty grows sharply from 16nm to 11nm mainly because below 11nm, the CMP is mainly **area bounded**, and since the area scaling is certain (Listing 3.1 Line 25), it limits the amount of uncertainty that gets propagated to the optimal core count. Meanwhile, when the tech node scales to 11nm and beyond, the CMP becomes **power bounded** and is extremely sensitive to the power uncertainties propagated from the uncertainty of the power scaling factor.

Figure 3.9 shows the actual functional graph generated by Charm. In terms of execution performance, we compare Charm execution to an unoptimized baseline in which all computa-

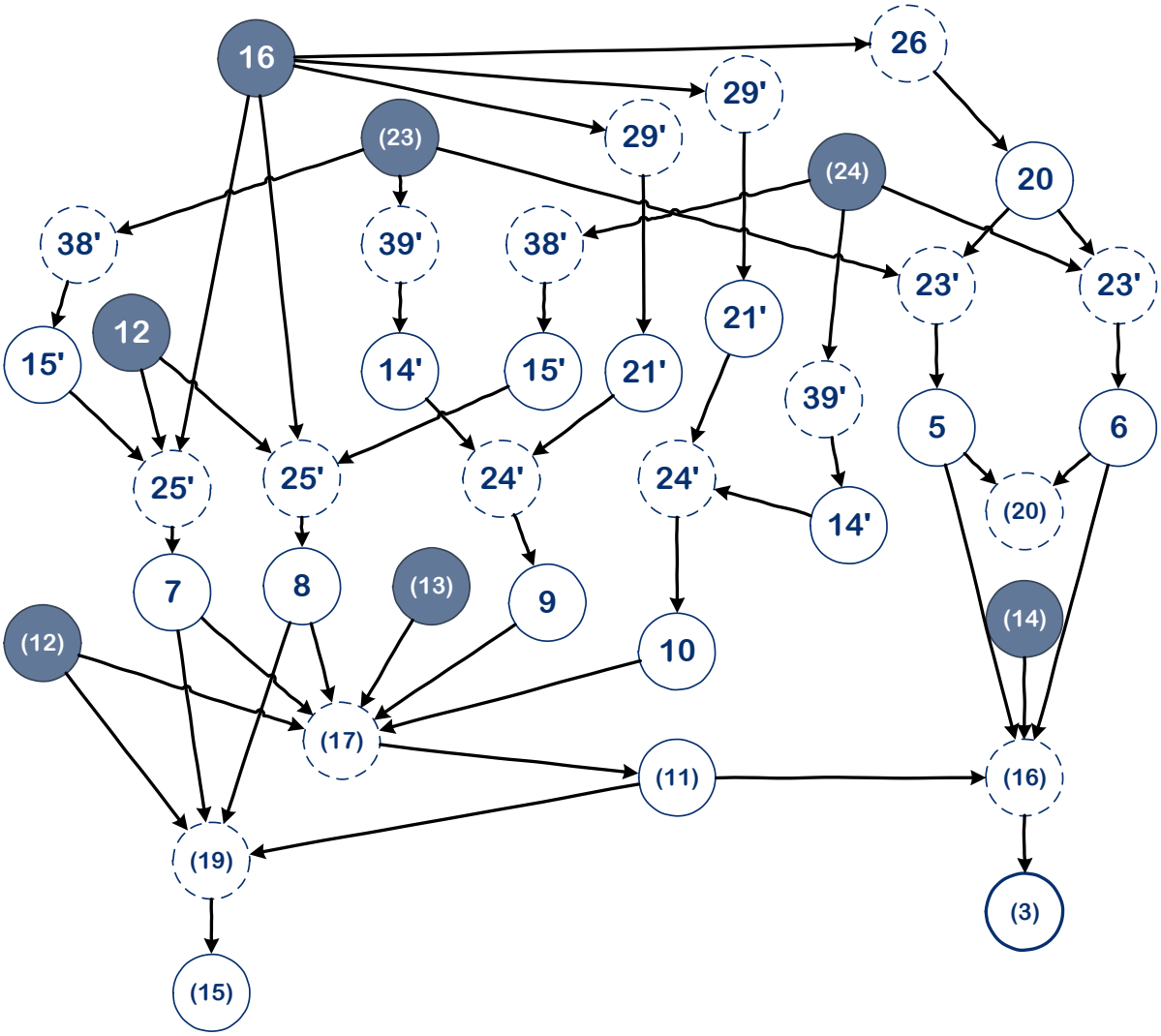


Figure 3.9: Functional graph generated by Charm for asymmetric dark silicon model. Node labels correspond to line numbers in Charm source code. As we present the asymmetric model code separately from the rest, plain numbers correspond to lines in Listing 3.1, and numbers in parenthesis correspond to lines in Listing 3.3. Numbers with prime (e.g., 21') are cloned names/equations generated by Charm from the same line of code.

tion is re-done per iteration (no invariant hoisting nor memoization). For ITRS or conservative scaling with asymmetric topology (a design space of 150K design points), full-blown Charm finishes on average within 120.5s, while the unoptimized implementation uses 159.5s (1.3X speedup). For the uncertain scaling with a MC sample size of 200 ($\tilde{1}$ million design points), optimized Charm uses 1562.5s, and it takes 5703.1s for the baseline implementation (3.6X speedup) on a single Intel i7 core at 3.3GHz to finish.

3.5.2 Surface Code Error Corrected Quantum Application and Architecture Co-optimization

In this section, a high level model for the resource overhead for implementing magic state distillation on surface code [92–94] is described and implemented within the Charm framework, which is used to pinpoint nontrivial interactions between fundamental system parameters.

For this study, we focus primarily on the Bravyi-Haah "$3k + 8 \rightarrow k$" procedure [92] augmented with the block-code protocol. By recursively stacking magic state distillation protocols in a tree-like fashion, one can generate arbitrarily high-fidelity magic states, which is required by a quantum program [94]. The space required by one round of Bravyi-Haah magic state distillation is given by the number of physical qubits required to run the circuit. Using block code, the procedure will consume $(3k+8)^{\ell-1}(6k+14)d^2$ physical qubits, where d is the surface code distance we are using.

Adding more factory capacity K results in more output magic state capacity (higher effective rate). However this also adds more components to the factory that may fail. In fact, a magic state factory has a **yield rate** proportional to the output capacity K that is caused by uncertainty in the success probability of the underlying Bravyi-Haah protocol. This yield rate

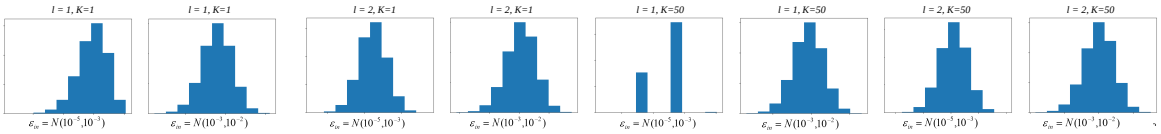


Figure 3.10: Factories designed with an implied low physical error rate only require concatenation up to $\ell = 1$ level, while more pessimistic factories require $\ell = 2$. While larger factories with $K = 50$ consistently show lower mean space-time consumption, they also suffer from large performance uncertainty when the assumed design error rate varies.

scales as:

$$K_{\text{output}} = k^{\ell} \times \prod_{r=1}^{\ell} \left[1 - (3k + 8)\epsilon_r\right] \tag{3.1}$$

where $\epsilon_r = (1 + 3k)^{2^r - 1}\epsilon_{\text{in}}^{2^r}$, because each level of the process results in incrementally higher fidelity (i.e., lower error rate).

Given a T gate request distribution D representing a program, the number of iterations needed to distill is:

$$\sum_{t=0}^{T_{\text{peak}}} \left(s \cdot \sqrt{K'} + \sqrt{\frac{t - sK'}{2}} \cdot R\right) \cdot L_{\text{cp}} \cdot D[t], \tag{3.2}$$

where $s = \left\lfloor \frac{t}{K'} \right\rfloor$, and $R = \frac{7d+15}{24d\ell}$. All of these equations combine to form a high level space-time estimate of the resources required to execute a quantum application on a machine with a specified magic state distillation factory architecture.

Using Charm, we are able to analyze the underlying sensitivity of different magic state factory architectures to variations in the underlying error rate of the physical system. We examine two different design cases, one where the factory is designed assuming a 10^{-3} error rate, and one assuming a 10^{-5} error rate. Figure 3.10 illustrates that while the time-optimal factory does show a lower expected space-time volume, it also shows significantly higher uncertainty and spreads in performance values over the space-optimal factory. This design point clearly moti-

vates that quantifying the uncertainty of a physical device is necessary to lead to risk-optimal system designs that perform well on a given system.

Charm is able to discover and quantify this trend with minimum effort, and allows for a quantitative analysis to be performed on these designs that will aid the construction of physical systems. Additionally, implementing this high level performance model in Charm allows for validation and more domain-specific error catching that previous implementations in Mathematica has been unable to catch. Specifically, a previous implementation has an incorrect parameter passed into a distance calculation function that Mathematica allowed to flow through. Charm is able to detect this error, warns that the models cannot be connected properly which helped correct the results of the model.

3.5.3 Exploration of CNN Tiling on FPGAs

In this case study, we explore an under-determined system with Charm. To evaluate CNN tiling on a set of Xilinx FPGA boards analytically, we take the analytical model from previous work [91] and use it to demonstrate the SAT-based searching capabilities of Charm. We let Charm automatically explore the configuration space for tiling and discover several new findings in this parameter space. We first describe the model for CNNs and the FPGA boards we are targeting. Then we ask the question: what is the optimal tiling configuration of a CNN architecture to maximize performance on a specific FPGA board?

CNN and FPGA models

Roofline model is an intuitive tool used to visualize a design's performance under the constraints of the platform's peak performance and maximum bandwidth [95]. To enable simultaneous multi-platform exploration, we allow an overlapping of a variety of "rooflines". Moreover, besides the compute and bandwidth ceilings defined by the platform's specifications,

Charm allows the user to set "floors": the lowest acceptable performance and the computation-to-communication ratio, which is directly related to the energy cost per operation.

For convolutional neural networks (CNNs), the convolution layers take up most of the computation time [**?**]. Many methods for efficient implementations of these networks on hardware have been proposed, building FPGA-accelerators being one of them. Zhang, et al. [91] build a roofline model based on memory bandwidth and logic resources with tiling:

$$\begin{aligned} computational\ roof &= \frac{total\ \#\ of\ operations}{\#\ of\ execution\ cycles} \\ &\approx \frac{2 \times R \times C \times M \times N \times K \times K}{[]\frac{M}{T_m} \times []\frac{N}{T_N} \times R \times C \times K \times K} \end{aligned} \tag{3.3}$$

$$\begin{aligned} CTC\ ratio &= \frac{total\ \#\ of\ operations}{total\ external\ data\ access} \\ &= \frac{2 \times R \times C \times M \times N \times K \times K}{\alpha_{in} \times B_{in} + \alpha_{wght} + \alpha_{out} \times B_{out}} \end{aligned} \tag{3.4}$$

where,

$$B_{in} = T_n(S\,T_r + K - S)(S\,T_c + K - S) \tag{3.5}$$

$$B_{wght} = T_m T_n K^2 \tag{3.6}$$

$$B_{out} = T_m T_r T_c \tag{3.7}$$

$$0 < B_{in} + B_{wght} + B_{out} < BRAM_{capacity} \tag{3.8}$$

$$\alpha_{in} = \alpha_{weight} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \tag{3.9}$$

$$\alpha_{out} = \frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c} \tag{3.10}$$

Here, the tiling design space consists of tile dimensions T_r, T_c, T_m and T_n. These param-

```
define xilinx_xc7vh870t_3:
    computation : R+ as cmpt
    computation_roof : R+ as roof
    bandwidth : R+ as bw
    bram_usage : R+
    roof = 3734.0
    cmpt <= roof
    bw <= 420.0/8
    bram_usage <= 50760.0 * 1000 * 1000/8
```

Listing 3.6: Hardware constraints from an FPGA board

eters are bounded by the corresponding dimensions of the NN structure. $\alpha_{in}, \alpha_{out}, \alpha_{wght}$ and $B_{in}, B_{out}, B_{wght}$ denote the trip counts and buffer sizes for accesses to the input feature maps, output feature maps, and the weights, respectively.

With the help of this analytic model, we can now explore the tiling configuration space given any platform in the roofline space. An example FPGA board (xilinx_xc7vh870t_3) in Charm is presented in Listing 3.6. The full Charm code for the CNN model is listed in Listing 3.7.

Optimizing Design Under Constraints

Here, we try to find an optimal configuration, in terms of a) performance, b) bandwidth requirement, and c) both performance and bandwidth requirement, by generating constraints on-the-fly while exploring the design space.

We use the CNN model of AlexNet [**?**]. AlexNet consists of 5 convolution and 3 fully-connected layers. For this experiment, we consider only the convolution part of this model. We look for the optimal (T_m, T_n, T_r, T_c) configuration for the second layer of the CNN model by dynamically adding lower-bound performance and/or upper-bound bandwidth constraints to make the SMT solver find a better configuration iteratively.

Figure 3.11 presents the search result. We can tell that the lower-/upper- bound constraints quickly guide the search to a more interesting area. By utilizing a state-of-the-art SMT solver,

```
define CNN:
    R : I+
    C : I+
    M : I+
    N : I+
    K : I+
    T_m : I+
    T_n : I+
    computation : R+ as comp
    comp = (2 * M * N) / (ceiling(M / T_m) *
        ceiling(N / T_n))

    a_in : R+
    a_weight : R+
    a_out : R+
    B_in : I+
    B_weight : I+
    B_out : I+
    T_r : I+
    T_c : I+
    S : I+
    bram_usage : R+ as bram
    computation_to_communication_ratio : R+ as ctc
    bandwidth : R+ as bw
    a_in = (M * N * R * C) / (T_m * T_n * T_r * T_c)
    a_weight = a_in
    a_out = (M * R * C) / (T_m * T_r * T_c)
    B_in = T_n * (S * T_r + K - S) *
        (S * T_c + K - S)
    B_weight = T_m * T_n * K * K
    B_out = T_m * T_r * T_c
    bram = B_in + B_weight + B_out
    bw = comp / ctc
    ctc = (R * C * M * N * K * K) /
        (a_in * B_in + a_weight * B_weight +
            a_out * B_out)

    T_m > 0
    T_n > 0
    T_r > 0
    T_c > 0
    T_m <= M
    T_n <= N
    T_r <= R
    T_c <= C
```

Listing 3.7: CNN roofline model

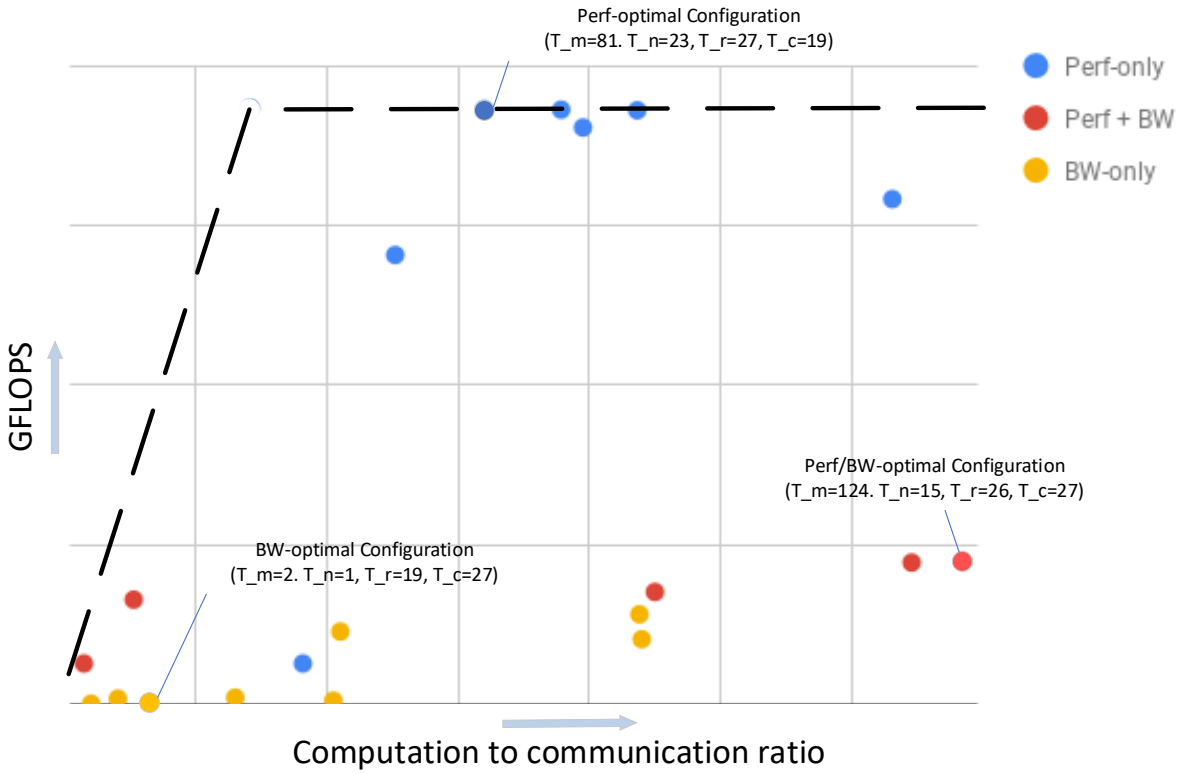


Figure 3.11: Finding optimal tile configuration for the second convolution layer of AlexNet under the roofline constraint when targeting xc7vh870t. "Perf-only" are the exploration trajectory when we only optimize for performance while "Perf + BW" shows the trajectory when we simultaneously optimize for both performance and the computation to communication overhead. In the first case, the SMT instance is iteratively bounded towards the upper part of the graph, i.e., the subspace with higher performance, while the second case pushes the search to the upper-right corner where designs have both better performance and a lower requirement on bandwidth.

without additional effort from the model builder, Charm greatly reduces the time spent on sweeping the design parameters. In the above evaluation, a brute force search over the space ($< T_m \times T_n \times T_r \times T_c >$, Line 38-45 in Listing 3.7) takes more than 5 hours to finish, while automatic exploration finds optimality in roughly 30 minutes, achieving more than 10x speedup. We expect this speedup to only grow as the model gets more complicated with more than 4 parameters to search for. Figure 3.12 transforms the roofline space into a performance-bandwidth space.

There are a few interesting observations from these two viewpoints of the space:

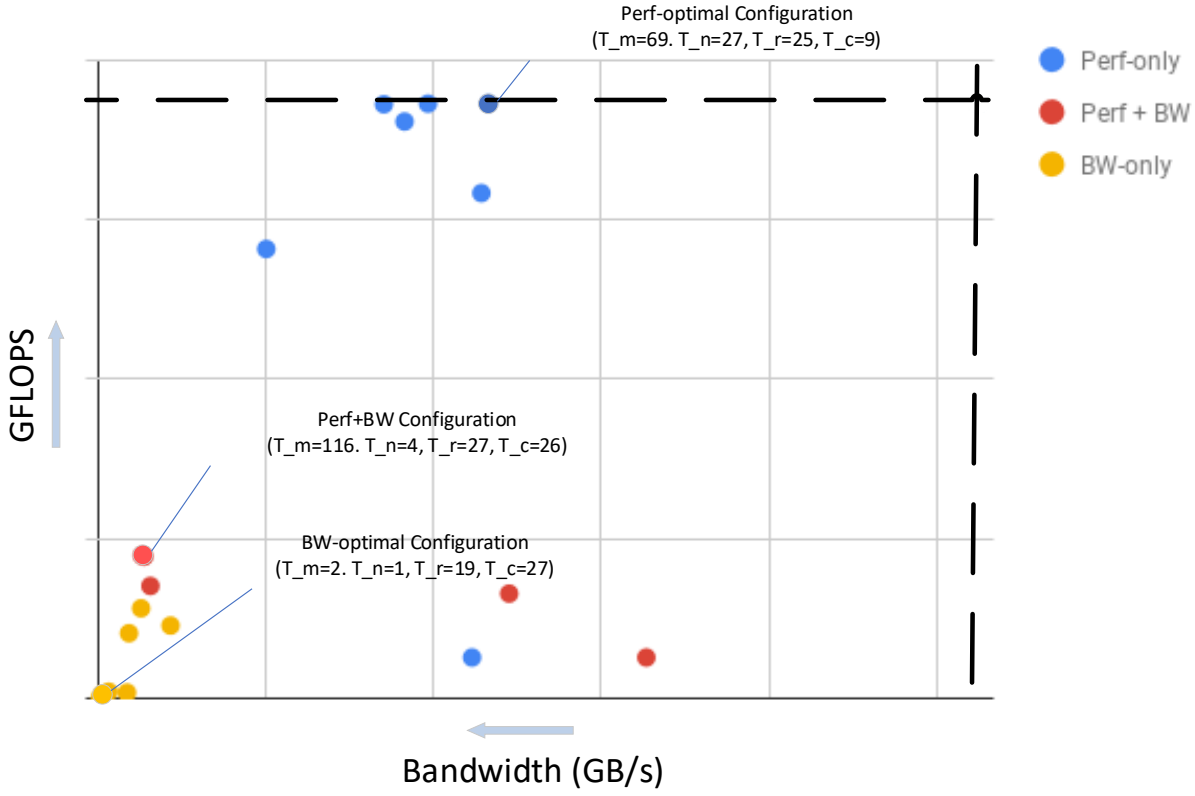


Figure 3.12: Projecting the search results into the performance-bandwidth space.

- From Figure 3.12, we can tell that performance does not have a clear linear correlation with bandwidth, i.e., to achieve better performance, a balance between processing resource and communication is more important rather than simply dumping logic transistors or using wide communication channels.

- The above said balance can be seen from the most performant few configurations. All these configurations are "asymmetric" and retain some ratio between T_m and T_n, (a possible interpretation maps to the on-chip processing resource and the communication channel width). As this ratio deviates from the "balance", performance degrades as we can tell from the configuration of "BW only" and "Perf + BW".

- From Figure 3.12 we can tell that this FPGA board is clearly bounded by its computation resources rather than communication bandwidth as designs within $> 99\%$ of the computation roof consume less than half of the available bandwidth.

3.6 Chapter Summary

Complex and intricately interacting constraints around energy, temperature, performance, cost, and fabrication create a web of relationships. As we move toward more heterogeneous and accelerator-heavy techniques, our understanding of these relationships is more critical for guiding the processes of design and evaluation than ever before. Already today we are seeing machine learning [96], cryptography [97], and other fields attempting to pull architectural analysis into their own work – sometimes introducing serious bugs along the way. Architecture is now a field that is expected to make scientific statements connecting nano-scale device details to the largest warehouse scale computers and everything in between. Spanning these 11 orders of magnitude will require more complex analytic approaches to be used in tandem with traditional simulation and prototyping tools that computer architects have long relied on.

Charm provides domain specific language support for architecture modeling in a way that leads to more flexible, scalable, shareable, and correct analytic models. While our language already supports symbolic restructuring, memoization, hoisting (and several other optimizations), consistency checks, and the capability of automatic exploration, Charm is merely the first step towards a more powerful and useful modeling language for computer architects. It is easy to imagine other useful additions in the future, such as checks on the consistency of physical types (e.g., nJ versus pJ errors) or back-ends connecting models to non-linear optimizers. Most importantly though, by giving the sets of mutually dependent architectural relationships a common language, Charm along with the collection of established models have the potential to enable more complete and precise specification, easier composition, more through checking, and (most importantly) broader reuse and sharing of complex analytic models.

Chapter 4

Accurate and Efficient Uncertainty and Risk Quantification with Detailed Simulation

While Chapter 2 and 3 define and explore uncertainty-induced effects in architecture design, i.e., architectural risk, through analytic high-level analysis, they are fundamentally limited by the high abstraction level of the analytic model used. In later design cycles, without detailed simulation results, we are still left unsure of how uncertainties in a real system design interact, manifest, and impact performance.

In theory, one can always measure uncertainty and architectural risk via Monte Carlo experiments with cycle-accurate simulators. However, a simple thought experiment will reveal that these methods do not scale well because the total simulation cost will quickly grow beyond measure with the introduction of even just a few tens of design parameters with a few hundreds of samples each. Understanding the interactions between these design parameters is critical to understanding the impact of uncertainty but the sheer size of the required search space and the cost associated to acquire data from detailed simulation is tremendous. Gathering data

and then interpolating them with various techniques (e.g. machine learning methods such as spline regression [98]) is certainly possible but they still require a significant random sample of the entire feature space to be simulated which translates to thousands of simulation runs to build a reasonably accurate predictive model (even assuming the complex models already have their hyperparameters well tuned). However when it comes to uncertainty and risk estimation, the point-wise prediction accuracy is not necessarily needed if one can capture the statistical characteristics (e.g., moments) of the resulting performance distribution.

In this chapter, we take on the task of investigating uncertainty and its impact on system performance, i.e., architectural risk, in realistic architecture designs with detailed simulation. Rather than random samples interpolated, we take an approach where observation of prior knowledge informs samples. Under such a model each sample is carefully chosen which means precious few are needed to build a reasonable architectural model. However, doing this in a way that is robust to noise and does not introduce bias is no easy task. To tackle this challenge, we exploit recent advances in the theory and practice of generalized Polynomial Chaos (gPC) [99, 100] to automatically build surrogate models, based on the prior assumption of how these uncertain parameters are distributed, to replace detailed simulation of the system. By trading off point-wise prediction accuracy, these generated models have easy-to-compute analytical forms and accurately capture the distributional statistics of the population (e.g., mean value and shape of distribution) which suffices in uncertainty and architectural risk estimation. We show such models are useful in the study of the impact of voltage uncertainties in a real chip-multiprocessor.

4.1 Unique Challenges with Simulators

When trying to quantify uncertainties and their impact on system performance with detailed simulators, we are faced with three major challenges:

The first challenge is how to properly handle the dependencies between different input parameters. For example, one can assume that under process variation, at the architecture level both on-chip L1 and L2 caches have a non-standard latency value. These values can be drawn from independent distributions if one wishes to study the scaling trends, but in a more realistic setting, we have to consider the dependency between them to reflect the real effect of process variation. In this setting, although we wish to study the architecture-level performance, we have to dig down to lower levels to faithfully capture the relationship between these architecture parameters (e.g., the aforementioned latency values).

The second challenge is that at the architecture level many parameters are inherently *discrete*. Considering the above example of cache latency, although physically the cache response time is a continuous value in the time domain, in digitized microprocessors it is quantized (or binned) into number of cycles. A small disturbance in the access time might end up pushing the latency in cycle count to another bin, causing a non-smooth performance function. Many machine learning methods (especially the ones relying on gradients to perform) may not suit well without applying complicated approximations (if such approximation exists) or need a large training set in such scenarios. Figure 4.1 shows an example of the non-smoothness caused by architectural-level quantization.

The third challenge comes from one well-known drawback of cycle-level simulation, that it runs orders-of-magnitude slower compared to native execution. A few seconds in native execution requires hours or days to simulate.

The general case total simulation cost (time-wise) can be expressed in Equation (4.1).

$$Total\_Time\_Cost = \frac{(\#\_runs \times Time\_per\_Run)}{Parallelism} \tag{4.1}$$

In most cases, one has limited parallelism (resources). Hence, to reduce the total time cost

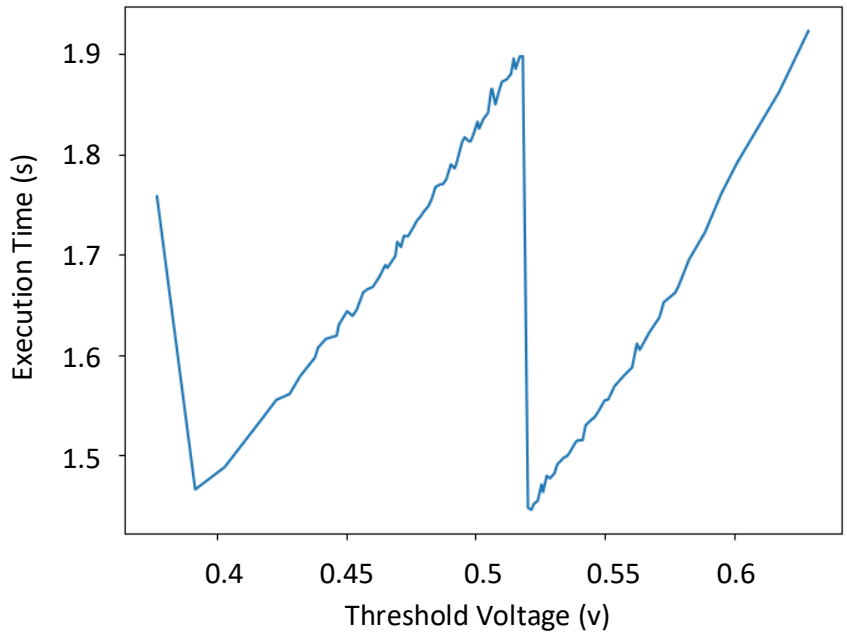


Figure 4.1: Non-smoothness due to architectural-level quantization. The execution time takes big 'dives' when the cycle latency moves from cycle N to cycle $N + 1$.

one has to utilize faster simulation techniques, reduce the number of total runs (i.e., reduce the coverage of the design space), or do both [101]. These techniques usually select a subset of instructions to simulate (via statistical sampling) to reduce time spent per run or reduce the number of programs/architectures/configurations evaluated (e.g., benchmark subsetting) to limit number of runs in total. All these techniques can be employed in our framework, but they do not directly address the new added dimensions from different uncertainties and are thus orthogonal to our method.

When uncertainty is under investigation, the number of simulation runs grows dramatically by an additional multiplier of $\#\_samples$. Sampling is used to capture the uncertain behavior of an uncertain input parameter. Taking one uncertainty as an example, the clock frequency of processor cores, to accurately estimate how it propagates and affects performance of the system, hundreds, even thousands, of samples are typically required in a Monte-Carlo experiment design to reconstruct a smooth distribution that approximates the ground truth closely. If one

wishes to study the convolution of multiple uncertainty sources to understand how they interact, the number of simulation increases exponentially with the number of uncertain variables and the number of samples when combinations of input variables are to be simulated. In terms of machine learning methods including regressions, it usually requires a training set that scales with the feature space which grows exponentially with the dimension of the input uncertainties. When it already requires hundreds or thousands of simulation to build the model without uncertainty [98, 102, 103], building such a model with uncertainty in the picture quickly becomes intractable.

All the above challenges hinder the investigation of uncertainty and architectural risk if not dealt with properly. We propose a general and scalable analysis framework exploiting multiple simulators at different levels and the advancement in the theories and practices of generalized Polynomial Chaos (gPC) expansion in Section 4.3 to overcome these challenges and enable efficient and accurate exploration. In particular, we make the following contributions:

Identifying the challenges in estimation of uncertainty/risk at the detailed simulation level. To the best of our knowledge, this work is the first to tackle the problem of estimating and analyzing uncertainties and architectural risks with a trusted, detailed simulator. We reveal the unique challenges of such a task including the need for analyzing dependencies among parameters, propagating uncertainties across different layers/simulators, the non-smoothness brought by this cross-layer analysis and the significant computational challenge for accurately capturing the statistical properties of propagated uncertainties and architectural risk.

Proposing a general framework to propagate uncertainties across different simulation levels. To properly model and propagate uncertainties across different abstraction levels, we combine both device- and architecture-level simulators to faithfully study how uncertainties at low level can impact system performance.

Enabling large-scale detailed exploration with small data. To address the mind-boggling

computational need to accurately estimate propagated uncertainties and architectural risk, we exploit the advance in both the theories and practices of generalized Polynomial Chaos (gPC) to efficiently build surrogate models with only a handful of simulation data. By utilizing the prior knowledge of how uncertainties are distributed, these surrogates provide accurate statistical estimations at the cost of losing point-wise predictability. As far as we know, we are the first to employ such a method to build surrogate models for architectural studies.

Quantifying the uncertainty and architectural risk of a realistic CMP system. Using our framework and the highly efficient model building methods, we explore a realistic CMP design under uncertainty through detailed simulations. To the best of our knowledge, this is the first thorough quantitative study of uncertainty and architectural risk of a realistic system at the cycle-accurate level. We find interesting new insights including which uncertainty source contributes more to the output performance variation and that, surprisingly, in some cases, reducing input uncertainty actually does not reduce the architectural risk the system is exposed to.

4.2 Related Work

4.2.1 Uncertainty Quantification and Sensitivity Analysis in Architecture Design

Uncertainty Quantification (UQ) has been an established process in many fields (e.g., finance, climate, biology, physics, circuit design) to identify and study uncertainties in the output of a model with regards to input uncertainties [104–108]. In terms of computer architecture and system design, Most of the existing work has been studying and quantifying individual uncertainty at different levels for different components in a system. Among this line of work, Borkar et. al. [9] identify and quantify the PVT variations and their impact on clock frequency

of a microprocessor. Bhardwaj and Vrudhula [109] study and quantify the impact of process variation on leakage at the circuit level. Zhang et. al. [110] quantify the uncertainty in leakage power consumption in a CMP under process variation. Wong et. al. [111] use analytic models to quantify the uncertainty in delay and power of FPGA devices under process variation. Das et. al. [112] quantify latency uncertainty in different micro-architectural units and propose cache mechanisms to improve batch performance under process variations. Yan et. al. [113] propose CoreRank to characterize and manage dynamic performance uncertainty introduced by silicon aging and degradation. Ozdemir et. al. [114] propose yield-aware architectural mechanisms for data cache to combat lower yield rate due to process variation. Liang et. al. [115] design a new 3T1D cache architecture to mitigate and mask process variation in caches. Regarding an all-around view of uncertainty propagation in system designs, Cui and Sherwood [19] use analytic models and Monte Carlo simulations to study the scaling trends and trade-offs of uncertainties in a CMP using a high-level analytic model. In this work, we, for the first time, enable and perform a thorough uncertainty quantification with detailed simulations.

Sensitivity Analysis (SA) is also widely adopted in many scientific fields for understanding model behaviors (i.e., impacts of uncertainty in the model inputs to the uncertainty in the model outputs) [116–118]. However, thorough sensitivity analysis is seldom applied in computer architecture and system design to the best of our knowledge. Fornaciari et. al. [119] applies derivative-based local sensitivity methods to guide analytic design space search for embedded systems. De Vogeleer et. al. [120] uses one-at-a-time sensitivity analysis to study the Energy/Frequency Convexity Rule with an analytic model for microprocessors. Zhu and Wong [121] apply sensitivity analysis on an analytic queueing model of a superscalar processor to study the relative importance of several micro-architecture configurations to performance. Arora et. al. [122] applies Plackett and Burman designs to search the complex design space of a multi-core processor. Their proposed method measures the impact of each parameter on the performance of the system instead of the variation of the performance. In this work, we take a

new perspective to provide a global sensitivity study for a CMP with a detailed simulator (as opposed to high-level analytic models) to capture the necessary architectural insights regarding how parameter uncertainties impact uncertainties of the performance of the system.

4.2.2 Surrogate Modeling in Architecture Simulations

A surrogate model is often used in place of slow simulators in many architecture studies. There are several kinds of popular surrogate models, including kriging methods (Gaussian process modeling) [123], artificial neural network (ANN) [102, 124], support vector machine (SVM) [125], linear (quadratic) response surface models [126], stochastic reduced-order models [127]. However, gPC-based method has rarely been used in architecture simulation. Generally speaking, a gPC model can achieve high prediction accuracy using only a small number of samples if the output function is smooth enough [128]. In our case, although the performance function is not entirely smooth, we find that the gPC model can still estimate the distributions with reasonably high accuracy. Zhang et. al. [129, 130] adopt the gPC-based model to do hierarchical uncertainty quantification for circuit analysis. Recently, gPC has improved to handle electronic systems with correlated or high-dimensional uncertain variables [131, 132]. Compared to gPC, other methods require a much larger number of samples to provide a good estimation.

Another whole class of research stapled as statistical simulations [43, 133] are also used for fast design space exploration. Oskin et. al. introduces a hybrid statistical/functional simulation methodology called ”HLS” [43]. By profiling the statistical characteristics of applications, the ”HLS” simulator symbolically executes a synthetic code sample to capture the performance of the architecture under study. Lee and Brooks use statistical inference and optimization to study efficiency trends for adaptive architectures [40]. Most of these works also focus on a different question of reducing systematic errors and achieve fast simulation speed and high design space

coverage, at the cost of accuracy. These methods, however, still require a relatively large amount of simulation work in order to build a useful surrogate model compared to our method.

In this work, we apply gPC method to build a highly efficient surrogate model with a very small number of samples. We also combine the stochastic testing method [134] to further reduce the number of samples to simulate.

4.3 A Cross-layer Scalable Analysis Framework with Surrogate Models

In this section, we propose a general framework that can efficiently and accurately estimate, propagate and evaluate uncertainties and architectural risk with a combination of detailed-simulators and gPC-enabled surrogate models. The key idea is rigorously picking samples based on the input uncertainty distribution to maximize the information coverage of the basis functions. We first briefly describe the overall workflow of our method followed by an elaboration of what and how generalized Polynomial Chaos (gPC) techniques can be used to build surrogate architecture models.

4.3.1 Modeling Workflow

Figure 4.2 visualizes the workflow of our analysis framework at a high level.

The input space to the experiment is described through a set of distribution specifications (for variables with uncertainties, i.e., they are random variables) and a set of fixed system configurations (for variables without uncertainties, i.e., they have fixed values). We first do a manual pass of refinement to identify/tease out uncertainties at different levels and distinguish dependent parameters from independent ones (e.g., cache latency is dependent on supply voltage and technology node). These independent distributions are then fed to the gPC workflow

described in Section 4.3.2 to generate a small set of sample points. We then automatically generate a set of device-level and architecture-level configurations from the training set and feed to the cycle-accurate simulator. The simulations are done from bottom up. When device-level simulation is finished, we rewrite the architecture-level configurations with the needed values from the device-level simulation results. Once we have the architecture-level results, we can extract the wanted statistics (e.g., simulation seconds and total processor energy consumption) and feed them back to the untrained model from gPC to determine the coefficients. The result of this final step is a fully trained surrogate model ready for use. In our analysis in Section 4.4, we use the model for uncertainty and risk quantification by running thousands of randomly generated input through the gPC model and we also extract sensitivity indices (e.g., Sobol's Indices [135]) directly from the built gPC model to estimate global sensitivity of each input variables as well as their higher-order interactions.

To validate the gPC model, we compare its predictions with the results from Monte-Carlo simulations. For that, we generate a large set of samples using Latin hypercube sampling [136] and feed them all to the cycle-accurate simulator. Then we feed the same sample set to the trained gPC models and validate the results of predictions and the statistics gathered from simulator instances. We present the validation results in Section 4.3.3.

4.3.2 Building Surrogates with gPC

To address the computational challenge brought by the input uncertainties, we employ an efficient spectral method, generalized Polynomial Chaos (gPC), to sample the vast input space in a much smarter way and to build surrogate models for each specific input space described by a set of distribution specifications (mean, standard deviation as well as lower and upper bounds) with merely a handful of training simulations required. We describe the theoretical background of gPC method at a high level, followed by a detailed elaboration of the steps to

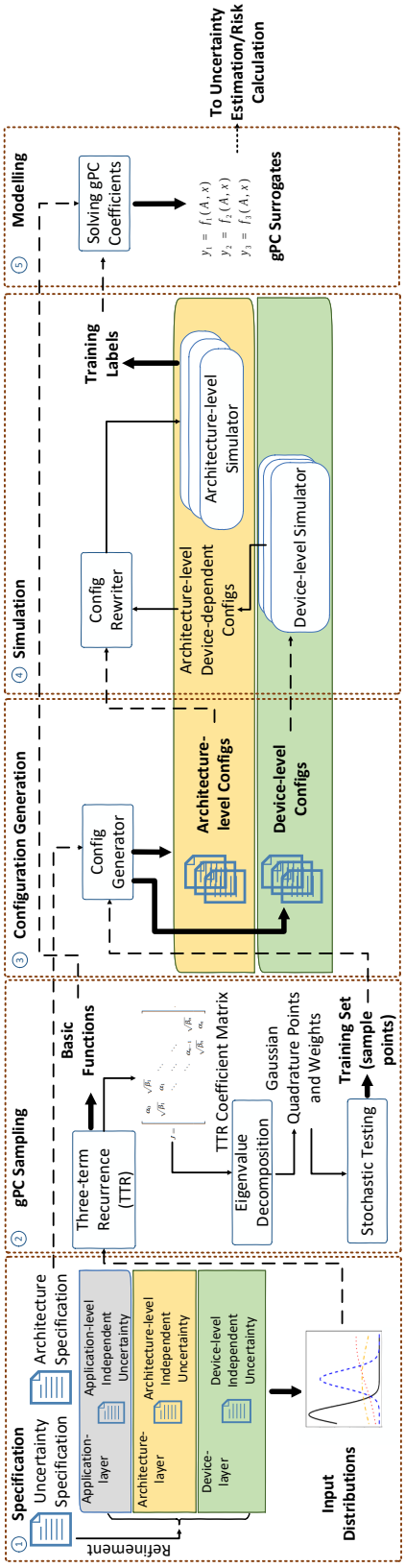


Figure 4.2: High-level overview of our analysis framework. We take the prior uncertainty specification and generate distributions that capture each independent source of uncertainty. These distributions are then fed to the gPC workflow which generates a set of parametrized basic functions and a set of training samples. We then generate a set of configurations with values from the training set and feed them to detailed simulators. Finally, we take the simulated stats and pass them along with the basic functions to build the gPC models which will serve as efficient and accurate surrogates for later exploration.

sample and build the model.

In this paper, gPC methodology is adopted as a surrogate model. The gPC model [128,137], which is an extension of the polynomial chaos (PC) to non-Gaussian cases, has become one of the most popular methods in uncertainty quantification [138, 139]. Let $\vec{x}$ be a d dimensional vector of stochastic variables, a gPC model is an approximation to the original function $y(\vec{x})$ as a weighted summation of a finite set of orthogonal basis functions in Equation (4.2):

$$y(\vec{x}) = \sum_{k=1}^{K} c_k H_k(\vec{x}), \tag{4.2}$$

where $H_k(\vec{x})$ is the k-th basis functions, c_k is the corresponding coefficient. K is the total number of the basis function, which is usually set as $K = \begin{pmatrix} p+d \\ p \end{pmatrix} = \frac{(p+d)!}{p!d!}$ in a popular total degree scheme [137], where p is called the order of gPC expansion. There are three major steps to build a gPC model: constructing basis functions, generating training samples and determining gPC coefficients.

Basis function construction: given a one-dimension uncertain variable with a probability density function of $\rho(x)$, in order to obtain $H_i(x)$ a set of orthogonal polynomials π_i are first constructed via a three-term recurrence (TTR) relation shown in Equation (4.3) [140]:

$$\begin{aligned} &\pi_{i+1}(x) = (x - \alpha_i)\pi_i(x) - \beta_i(x)\pi_{i-1}(x), \quad i = 0, 1, \cdots, n \\ &\pi_{-1}(x) = 0, \quad \pi_0(x) = 1, \end{aligned} \tag{4.3}$$

where

$$\alpha_i = \frac{E\left[x\pi_i^2(x)\right]}{E\left[\pi_i^2(x)\right]}, \quad \beta_{i+1} = \frac{E\left[\pi_{i+1}^2(x)\right]}{E\left[\pi_i^2(x)\right]}, \quad i = 0, 1, \ldots, n \tag{4.4}$$

and $\beta_i = 1$. Then the orthogonal polynomials can be obtained via normalizing the obtained $n + 1$ basis functions with Equation (4.5):

$$H_i(x) = \frac{\pi_i(x)}{\sqrt{\beta_0\beta_1\cdots\beta_i}}, \quad i = 0, 1, \ldots, n. \tag{4.5}$$

Notice that the obtained basis function is univariate. However, generating multivariate basis functions is trivial: we can use choose one univariate basis function for each uncertain variable, then their product generates a multivariate basis function [129].

Training set generation: In this paper, we use a stochastic testing (ST) method [134] to generate the training set in order to determine the coefficients later. With the ST method, a total number of $(p + 1)^d$ Gaussian quadrature points are generated first as the candidate points by applying the Gaussian quadrature rule (GQR). GQR approximates the definite integral of a function as a weighted sum of function value of specific points (i.e., Gaussian quadrature points). Given a specific density function $\rho(x)$, Gaussian quadrature points x^j and their weights w^j are fixed. With the basis of TTR relation [Equation (4.3)], we first generate a TTR coefficient matrix, which is a symmetric tridiagonal matrix defined in Equation (4.6):

$$J = \begin{bmatrix} \alpha_0 & \sqrt{\beta_1} & & & \\ \sqrt{\beta_1} & \alpha_1 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1} & \sqrt{\beta_n} \\ & & & \sqrt{\beta_n} & \alpha_n \end{bmatrix}. \tag{4.6}$$

With eigenvalue decomposition, denoted as $J = U\Sigma U^T$, we can easily calculate quadrature points and weights: x^j is the j-th diagonal element of Σ, and its corresponding weight w^j is $u_{1,j}^2$ [140]. Similarly, we can also easily extend the univariate GQR to multidimensional cases [141] to obtain multidimensional Gaussian quadrature points.

Then a set of K ($K \ll (p+1)^d$) sample points, called ST points, are selected from the obtained candidate quadrature points as a training set of the gPC model. The ST method selects K samples based on two criteria: (1) it chooses important samples with large quadrature weights; (2) it seeks for new sample points which can extend the vector space of basis functions [134]. Figure 4.3 shows an example of the selected ST points in our experiments in Section 4.4.

gPC coefficient determination: upon getting the training labels from the simulation, we solve for the coefficients through a set of deterministic equations (4.7):

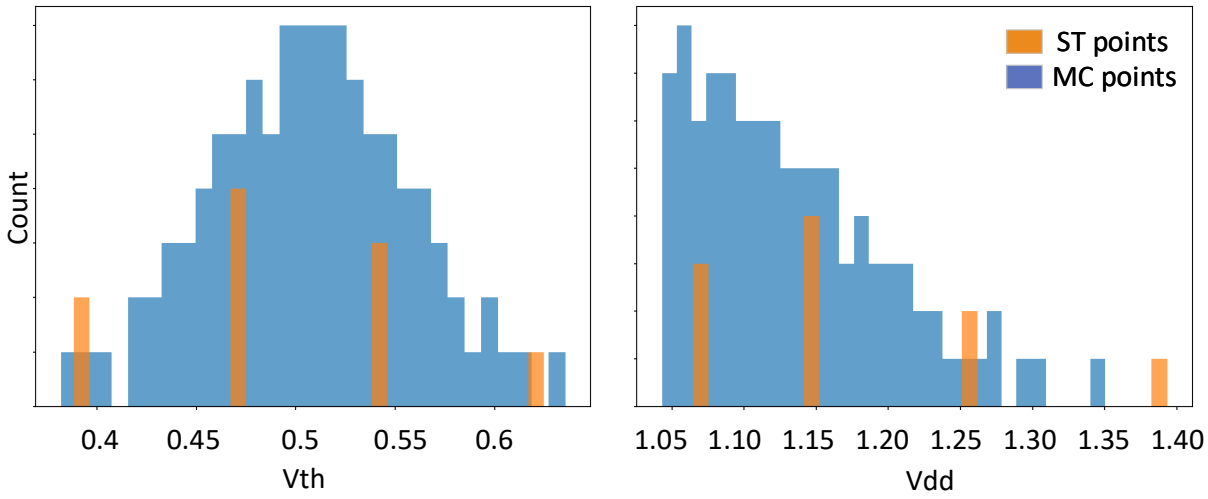


Figure 4.3: ST points picked by the GQR among the population of 100 randomly selected Monte-Carlo points for both V_{th} and V_{dd}. These ST points are reasonably distributed to cover the entire input space.

$$\begin{cases} y(\vec{x}_1) = \sum_{k=1}^{K} c_k H_k(\vec{x}_1) \\ y(\vec{x}_2) = \sum_{k=1}^{K} c_k H_k(\vec{x}_2) \\ \vdots \\ y(\vec{x}_K) = \sum_{k=1}^{K} c_k H_k(\vec{x}_K) \end{cases} \tag{4.7}$$

4.3.3 Validation of gPC Models

Here, we present the validation results of the order-3 gPC model we used in Section 4.4 at one specific input uncertainty level (10%) and technology node (40nm) against 100 Monte-Carlo data points to demonstrate the accuracy of such surrogate models. We also compare the gPC estimator with a few other popular regression techniques including linear regressor, support vector regressor (SVR) and gaussian process regressor (GPR).

Figure 4.4: Validation of surrogate models with a gPC order of 3 with 10% uncertainty in V_{th} and V_{dd} at 40nm technology. The top set shows the comparison of histograms and statistics of gPC model predictions and the ground truth Monte-Carlo results (MC). The second set shows that of linear regressor and MC. The third set has gaussian process regressor (with GBF kernel) and MC, while the last set compares support vector regressor (with polynomial kernel) and MC. All the regression techniques are trained using the same number of samples as the gPC training set but evenly picked from the input space.

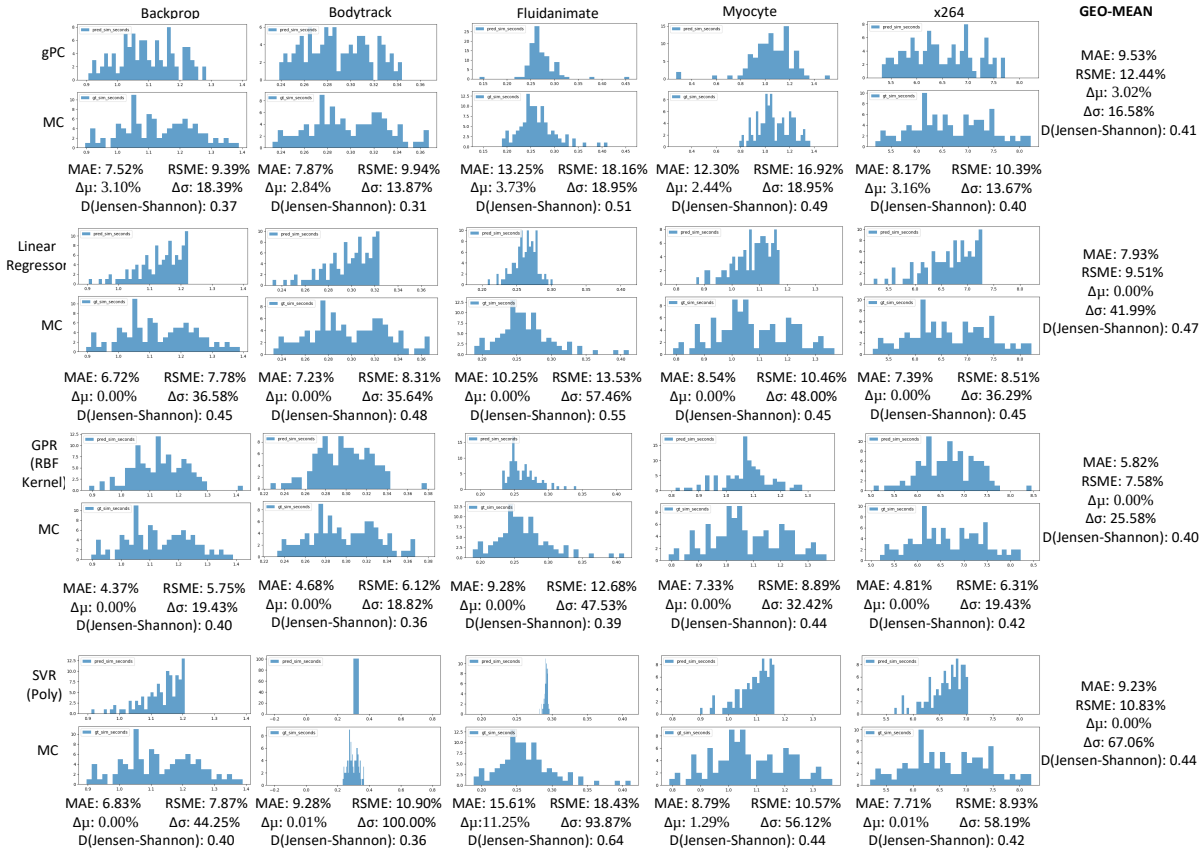


Figure 4.5: Comparing gPC estimators and theoretical limits of linear regressor, support vector regressor and gaussian process regressor. gPC models are still trained with merely 10 samples while all other regressors are trained with the exact test set, i.e., 100 Monte-Carlo samples, which should work in their favorite and serve as an upper bound of how well these regressors can perform on this test set.

Figure 4.4 presents the validation results along with comparisons against a few popular regression techniques one might use to build such a surrogate model. As we can tell from the histograms, with as few as 10 samples in this case, gPC surrogates accurately capture the position, scale and even shape of the ground truth distributions, which is crucial when we estimate propagated uncertainties and calculate architectural risk later. For all other regressors, it appears that they cannot closely estimate the ground truth, at least not with a training set as small as 10 samples (for 10 reasonably distributed samples, with random samples, the statistics are far too unstable to serve as an estimator). If we look at the geometric means of the statistics, we notice that, as expected, the gPC surrogates don't show the best point-wise predictions, as captured by mean absolute error (MAE) and root mean squared error (RMSE), but they mostly accurately provide a stable estimator for the moments (mean value captured by $\Delta\mu$ and standard deviation captured by $\Delta\sigma$) as well as the shape of the distribution (having the lowest Jensen-Shannon distance between the predicted histograms and the ground truth).

In light of the bad performance of the other regressors with 10 samples, we perform another comparison between gPC models (with 10 samples) and the theoretic limits of the other regressors trained and tested on the same set (100 Monte-Carlo samples). Figure 4.5 presents the results. As we can tell from the statistics, all regressors have better point-wise prediction power as expected. Interestingly, except for GPR which has an on-par performance with gPC model in terms of capturing the higher-order moments (standard deviation in this case) and the shape of the distribution, both linear regressor and SVR still have a worse estimation of the distribution itself. When estimating architectural risk, accurately capturing the distribution itself (not only just the mean value) is absolutely critical as the risk is directly computed from the distribution.

4.4 An Analysis of Uncertainties in A Chip-multiprocessor Architecture

In this section, we apply our framework to study the impact of uncertainties from low-level voltage parameters to chip-multiprocessor performance at the architecture level. We first list in detail all our configurations and assumptions. We then validate the surrogates by comparing a variety of metrics against that of the Monte-Carlo experiment results along side three other popular regression methods typically used to build predictive models from a data set. Lastly, we use these models to study the behavior of uncertainties and architectural risks in the system when the input uncertainty level changes.

The System Under Study

Here we first clearly scope our study in this section by defining the system under investigation and the uncertainty models we use in evaluation.

Architecture and Configuration. We investigate a homogeneous CMP similar to the baseline system used in the NX3T system [142]. Figure 4.6 shows the architecture of the system.

Table 4.1 lists all configurations that are fixed in the architecture, i.e., we will not injecting any uncertainty in these configurations. Changing these knobs are interesting if one is particularly looking at a different design options. But here, we distinguish design configurability from uncertainties. Scaling our analysis to include different design options is trivial but it will make interpreting the results more complicated.

The architecture we choose is a design that has been studied extensively. Therefore, it is interesting to see if even with this mature design we can still discover new understandings with uncertainty and architectural risk in the picture.

Uncertainty Parameters and Models. There are many parameters that are susceptible to any

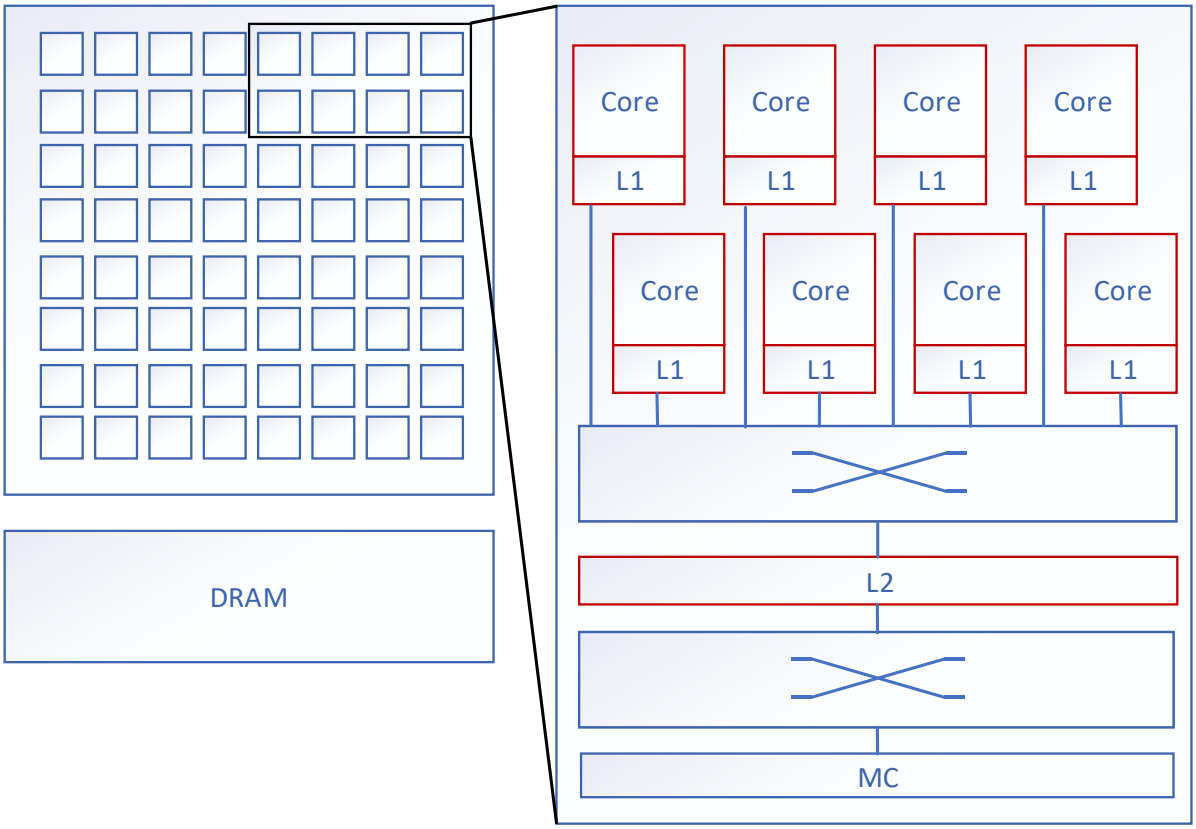


Figure 4.6: The system architecture of interest and the components that are affected by uncertainties in our case study. Each 8 cores share an L2 cache. We consider uncertainties propagated from the device level to each of the 64 cores, latency of L1 caches and latency of L2 caches.

one of, or a combination of, projection, process and design uncertainties. For simplicity and to limit the complexity of interpreting the results, we only consider the two most prominent and important uncertainty sources in this work: the variations in threshold voltage and supply voltage. Uncertainty in supply voltage could come from process variation while that in threshold voltage can be from both process variation and projection uncertainty. Due to limitations of simulation infrastructures, we also only consider die-to-die (D2D) process variation in our case study here. Table 4.2 lists the the uncertainty model (distribution) we use.

For both V_{th} and V_{dd}, we model them with truncated Gaussian distribution after existing

Table 4.1: Fixed system configurations.

Component	Configuration	Notes
# cores	64	8-issue, in-order, Alpha ISA
L1 I-cache	32KB, 2-way	private per core, 64B linesize
L1 D-cache	64KB, 2-way	private per core, 64B linesize
L2 cache	4MB, 8-way	shared per 8 cores, no prefetcher, 64B linesize
On-chip XBar	8B-wide	coherent, snooping protocol
DRAM	DDR3-1600	

Table 4.2: Uncertainty models for V_{th} and V_{dd}.

$$V_{th} \sim Truncated\_Gaussian(\mu, \sigma, 0) \tag{4.8}$$
$$V_{dd} \sim Truncated\_Gaussian(\mu, \sigma, \alpha) \tag{4.9}$$
$$V_{th} < V_{dd} \tag{4.10}$$

work [10]. The mean values μ in Table 4.2 comes from the nominal threshold voltage and supply voltage at the chosen technology node from the ITRS roadmap. The lower bound of supply voltage is limited by what CACTI allows, i.e., we do not consider near-threshold voltage in this study. We control the level of uncertainty injected by assigning different values for the standard deviation, σ. The standard deviation is always interpreted as a percentage of the mean value of the corresponding distribution.

Simulation Setup

We use gem5 [17] to model the system architecture. To propagate uncertainties in voltages to the architecture level, we generate a series of configurations with different voltage values and feed them to CACTI [143], which computes the access time of each component we consider. Using the same set of voltages, we also derive the clock frequency of the die with Equation (4.11) [144].

$$CK = \frac{1}{F} \propto \frac{V_{dd}}{(V_{dd} - V_{th})^{1.3}} \tag{4.11}$$

After we derive the clock frequency, i.e., cycle time for the core, we then quantize the cache latency into cycle delays by dividing the access time by the clock cycle time.

We use a collection of 5 workloads from PARSEC [145] and Rodinia [146] benchmark suites to include some application level variations in terms of memory access patterns and different workload balance among the cores. All our experiments in Section 4.4.1 and Section 4.4.2 consist of 4 different input uncertainty levels, from a conservative setting of 10% (less than already observed [147]) to a more radical speculation of 50%, at 3 different technology nodes (22nm, 40nm and 65nm).

4.4.1 Exploring Architectural Risk at Different Input Uncertainty Levels

In this section, we generate 1000 Monte-Carlo points from the input distributions for each experiment and feed them to the surrogate models to explore the propagated uncertainties and architectural risk when the input uncertainty level changes and when we shift between different technology nodes.

Figure 4.7 shows the average-case performance (i.e., mean value) and the propagated amount of uncertainty in performance (execution time). Note that the average-case performance is normalized to the nominal value, i.e., the case when there's no uncertainty at all and both V_{th} and V_{dd} are at their nominal values. The propagated uncertainty is measured in absolute execution time (seconds) so that it is consistent with the scale of the input uncertainties. The general trend at each technology node when the system is exposed to more uncertainty matches our expectation that the propagated uncertainty grows and the average-case performance lowers. What's interesting is that the propagated uncertainty grows sub-linearly as the input uncertainty increases for older technology nodes (40nm and 65nm) even though the uncertainty at these nodes have larger variations in terms of absolute voltage values than that of 22nm. This suggests that such a CMP in newer technologies is more sensitive, or has less tolerance, to input voltage uncertainties. In an actual design, if one cannot eliminate uncertainty entirely, for older technologies it is probably enough just to confine the uncertainty in a certain range (50% in this case) without the need to trim it down to a very small amount (say 10%). In other words, fighting uncertainties with valuable engineering resources might have diminishing returns at these technology nodes. However, for 22nm, the propagated uncertainty grows super-linearly, and limiting the amount of uncertainty that goes into the system becomes more important.

In terms of the architectural risk the system is exposed to, we evaluate two risk functions in this experiment: a quadratic risk function that penalizes chips with low performance and

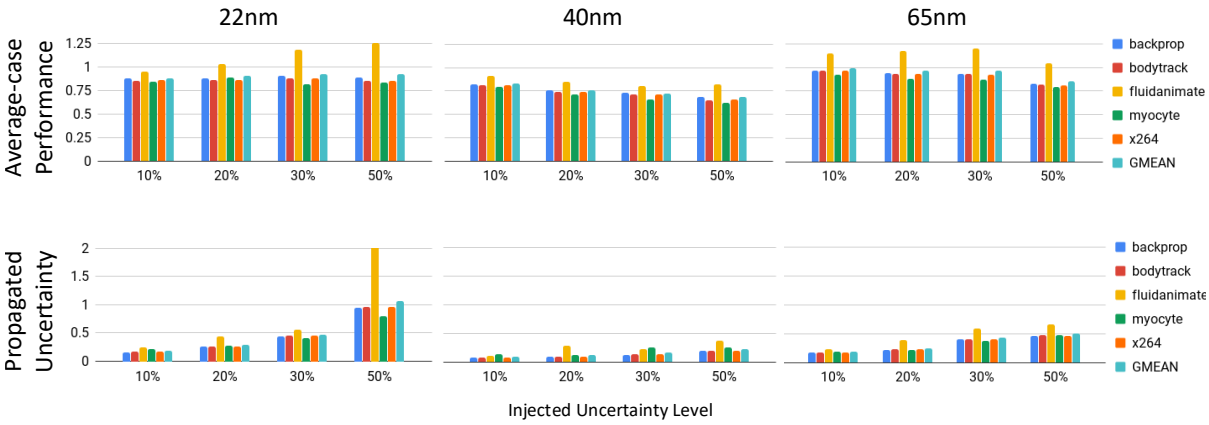


Figure 4.7: Average-case performance and propagated uncertainty with increasing uncertainty levels at different technology nodes. Average-case performance is the mean value of the performance distribution of the CMP while the propagated uncertainty level is captured by the standard deviation of the performance distribution.

a dollar risk function that maps the performance to binned processor sales price (based on an Intel CPU price list similar to [19]). Figure 4.8 shows the amount of architectural risk when we run different workloads at different uncertainty levels. One interesting observation here is that different risk functions behave very differently under uncertainties: the quadratic risk grows as the input uncertainty grows (pretty much follows the propagated uncertainty level) while the dollar risk is more complicated and random. Although the general trend for each technology node is that it still grows with the input uncertainty level, when we compare the amount of risk across different tech nodes, it does not show any clear trend. This makes quantifying architectural risk with our method more valuable since this information will help designers and managers decide whether to devote scarce engineering resources to eliminating uncertainties at all. For example, if the dollar value of a batch of chips is the sole concern then 50% input uncertainty actually does not do much worse than only 10% input uncertainty.

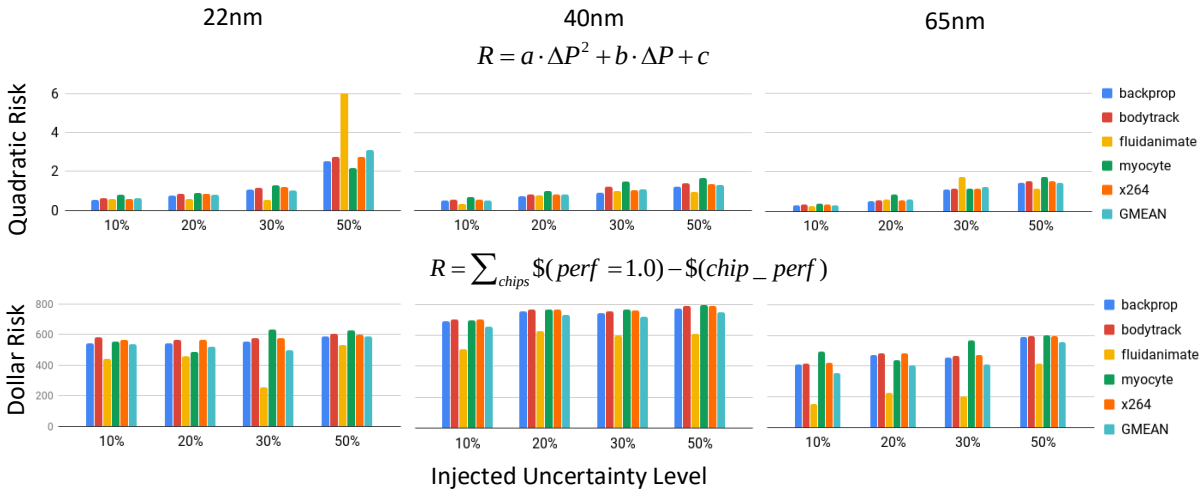


Figure 4.8: Two types of architectural risk with increasing uncertainty levels at different technology nodes. Quadratic risk puts more weight on the higher-order performance loss (a ¿ b), while dollar risk is the amount of dollars lost due to chips under-performing using a mapping function between execution time and sales price.

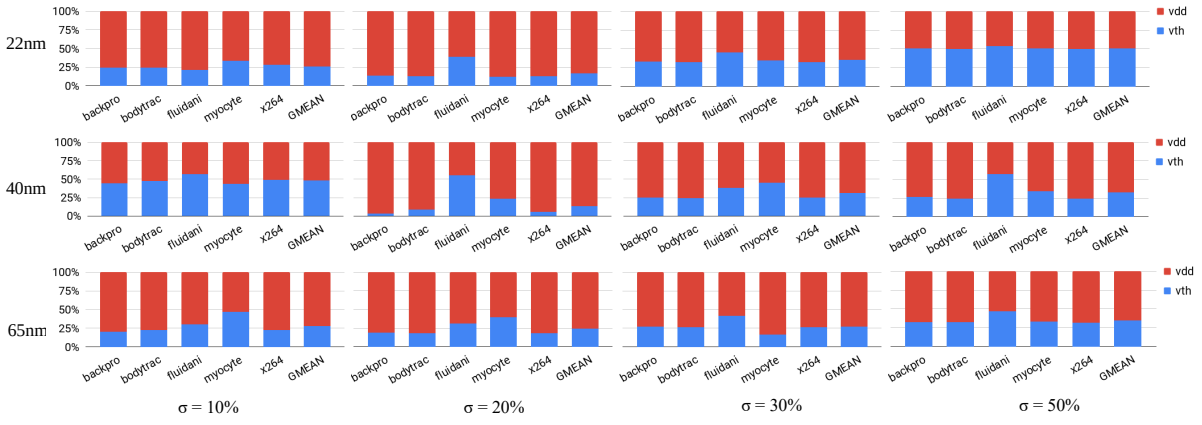


Figure 4.9: Proportion of total sensitivity of V_{th} and V_{dd}. The total sensitivity index captures the contribution to the output variance (uncertainty) from all terms that contains a certain parameter including the interaction terms with other input parameteres.

4.4.2 Global Sensitivity Analysis

In this section, we demonstrate another big advantage of using gPC-based surrogates: the analytical form of the built model allows direct computation of several interesting statistics. We specifically look at the global sensitivity of performance against the input uncertain parameters which can shed some light on the question of "which parameter causes more uncertainty in the output performance?" We can compute Sobol's indices [148], both main (M_i) and total (T_i), where i is a specific parameter we care about, using Equation (4.12) directly from the model in the form of Equation (4.2).

$$M_i = \frac{\sum_{S_k=i} c_k^2}{VAR(y)}, \quad T_i = \frac{\sum_{i \in S_k} c_k^2}{VAR(y)}, \tag{4.12}$$

where S_k = {uncertain parameters in H_k}.

Figure 4.9 plots the total sensitivity indices for the two input voltage parameters. We can tell from the figure that in most cases, no matter how much uncertainty is there in the inputs, V_{dd} contributes the most to the output uncertainty. This calls for active power management of an architecture in order to mitigate the performance hit introduced by process variations in the power supply circuit. Another interesting observation based from the figure is that, at 22nm, there's a clear trend of increasing uncertainty contribution from V_{th}. This suggests that upon designing new systems, or systems with newer technologies, an accurate projection of the threshold voltage and a precise estimation of the process variation in the device itself becomes more important. Once taped out, the uncertainty map of the threshold voltage can become the limiting factor if one tries to reduce uncertainties or risks in the final silicon chips.

4.5 Chapter Summary

In this chapter, we demonstrate an accurate and efficient way of quantifying and thus understanding uncertainties through surrogate models trained with detailed simulation results. We address the computational challenge with the power of generalized Polynomial Chaos expansions. With only a handful of data, we are able to build accurate and fast analytical surrogates that allow us to explore the manifestation and interaction of uncertainties and estimate the risks that they incur. With a mature CMP architecture, we acquire interesting new findings through such a study. The new capability of accurately and efficiently estimating uncertainties and risks, as well as a fully automated evaluation workflow and tool chain which we deem as a valuable future work, will become an important trick up in our sleeves. With that we believe computer architects will soon be able to reason about uncertainties and design systems that is more stable in the face of unknowns, more trusted with different performance constraints and more efficient when conservative margins are replaced with tailored designs.

Chapter 5

Conclusions

Computer architecture is entering a new era with a bounty of opportunity for innovation via emerging technologies. However, along with those opportunities are uncertainties which, if not dealt with carefully, might undermine all of our hard work. Between applications and the underlying physical circuits computer architects are in the best place to understand how those uncertainties will manifest at the system level. From this understanding we can take a more active approaches to combat the potential issues brought by them.

In this thesis, we thoroughly investigate how we should define and quantify different uncertainties in early system design process. In Chapter 2, we present a discussion of what uncertainties an architecture design is exposed to, how we can mathematically describe uncertainties and define their impacts, as well as a demonstration of the type of analysis we wish to perform and the new discoveries we are able to acquire from such an analysis.

This type of analysis essentially opens the door to a new architecture research space centered around three basic questions: **a)** how can architectural choices strike an effectively balance between performance, resource utilization (energy or area), and risk? **b)** how can one introduce new micro-architectural support, new computer organizations, and hardware design tools to change the risk-performance trade-off space in fundamentally new ways? and **c)** how

can we work together as a community to properly leverage the best thinking on each of these individual risk factors to create a cohesive understanding. While (a) and (b) are already a close parallel to other computer architecture work, (c) will require both new methods and coordination.

We take first steps in Chapter 3 and Chapter 4 proposing systematic ways to support such analysis and promote collaboration in early design time, for closed-form analytical models and detailed simulations respectively.

In Chapter 3, we start from the pain points when performing analytical analysis in an ad-hoc fashion and argue that a more systematic way is needed to support and scale such analysis. We take a programming language inspired approach by designing a custom modeling language to address the various issues with ad-hoc modeling. Through three case studies, we demonstrate how using Charm provides a variety of benefits.

In Chapter 4, we demonstrate an accurate and efficient way of quantifying and thus understanding uncertainties through surrogate models trained with detailed simulation results. We address the computational challenge with the power of generalized Polynomial Chaos (gPC) expansions. With only a handful of data, we are able to build accurate and fast analytical surrogates that allow us to explore the manifestation and interaction of uncertainties and estimate the risks that they incur. With a mature CMP architecture, we acquire interesting new findings through such a study.

5.1 Future Work

Looking forward, the convolution of uncertainties from all over the places in application, technology and system organization imposes a great challenge when we try to design better systems for the future. We believe the intersection of architecture, circuit design, algorithms, programming languages, formal methods and statistics is the right place to inspect, study, and

understand the interactions and impacts of all these uncertainties and to think about what can we do about them. Especially now that we are moving towards less explored design spaces like accelerators, 3D architectures, and even quantum computers, a joint analysis of all the above domains should be more widely adopted in the system design process in the future, as opposed to today when it is more of an ad-hoc practice (e.g., a "back of the envelope" estimation). As we demonstrated in this thesis, a joint analysis of risk assessment and performance through statistical sampling and simulation reveals new insights about how robust the performance of a design is in the face of uncertainties. This joint analysis (along with the need for a properly designed tool chain to support it) opens up unique and exciting research opportunities to bring concepts, techniques and constraints from all related domains together to enable new analysis, inspire new insights and develop new tools.

From this thesis, we see two future directions towards the realization of above said future design process: 1). to systematically support exporting and evaluating uncertainties across different levels (e.g., analytic models, functional simulation and RTL simulation), and 2). to innovate new methods, algorithms and architectures to mitigate and even exploit the influences of uncertainties across the software-hardware stack. It is unclear, and thus worth researching, that if either incorporating hardware mechanisms to transparently hide uncertainties from software (guard-banding being one example of this flavor) or exposing uncertainties to software (uncertain$\langle T \rangle$ [149] as an example of this type of solution) is the right way to mitigate uncertainties.

To Systematically Export and Evaluate Uncertainties Across Different Levels

The combination of programming language techniques and architectural analysis, like Charm, provides an unique opportunity to bridge high-level uncertainty analysis with lower-level detailed simulations. One way of automating this process is to formally express the input structures of commonly used architecture simulators (e.g., gem5) and extend Charm to automatically synthesis simulator instance which meets high-level uncertainty description and is ready

to execute by generating required input signals based on the high-level study result and the formal specifications of the needed inputs. Conversely, from bottom-up, through building an analytic model library for commonly used design patterns (similar to McPAT), it is also interesting to examine how to automatically compose an analytic model in Charm from a detailed design (e.g., in RTL or HLS languages) to assist design space exploration and cross-validation of the design (e.g., whether the uncertainty range checks with the high level description). With the abstraction of Charm and other possible extensions (e.g., integrating with HLS languages), experts from different domains can talk within the same intellectual framework to reason about architectures. With the help of the tool chain that comes along, this analysis procedure will also be easily accessible and sharable.

To Mitigate and Exploit Uncertainties across the SW/HW Stack

Ultimately it is worthwhile to rethink how we should build our microprocessors, memories and systems with a rigorous understanding of uncertainty in mind. It is promising to both leverage the existing techniques, most of which deal with one certain type of uncertainty, to propose new solutions with a stronger statistical guarantee, and also innovate new architectural tweaks and designs (e.g., utilize the fact that multiple uncertainties might cancel each other to a certain degree) to mitigate the impact and still maintain a strong average-case performance.

Further along, it will be very interested to investigate and build connections between uncertainty-aware system design with other domains of inherent variability, including approximate computing techniques (since approximation is likely to bring uncertainty to the system as a side-effect and conversely is tolerant to a certain amount of uncertainty) and security of a system (one example is the uncertain nature of SRAM may leak secret through start-up time statistical characterization [150]).

Bibliography

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et. al.*, *In-datacenter performance analysis of a tensor processing unit*, in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, IEEE, 2017.

[2] A. Madhavan, T. Sherwood, and D. Strukov, *Race logic: A hardware acceleration for dynamic programming algorithms*, in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 517–528, IEEE Press, 2014.

[3] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, *The missing memristor found*, *Nature* **453** (May 01, 2008) 80–3. Copyright - Copyright Nature Publishing Group May 1, 2008; Last updated - 2014-03-19; CODEN - NATUAS.

[4] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, *Spectre attacks: Exploiting speculative execution*, *arXiv preprint arXiv:1801.01203* (2018).

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown*, *arXiv preprint arXiv:1801.01207* (2018).

[6] M. Miranda, *The threat of semiconductor variability*, in *IEEE Spectrum*, June, 2012.

[7] K. Agarwal and S. Nassif, *Characterizing process variation in nanometer cmos*, in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, (New York, NY, USA), pp. 396–399, ACM, 2007.

[8] S. Mittal, *A survey of architectural techniques for managing process variation*, *ACM Comput. Surv.* **48** (Feb., 2016) 54:1–54:29.

[9] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, *Parameter variations and impact on circuits and microarchitecture*, in *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, (New York, NY, USA), pp. 338–342, ACM, 2003.

[10] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, *Varius: A model of process variation and resulting timing errors for microarchitects*, *IEEE Transactions on Semiconductor Manufacturing* **21** (Feb, 2008) 3–13.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, *The parsec benchmark suite: Characterization and architectural implications*, in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October, 2008.

[12] A. Limaye and T. Adegbija, *A workload characterization of the spec cpu2017 benchmark suite*, in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 149–158, April, 2018.

[13] A. Jaleel, *Memory characterization of workloads using instrumentation-driven simulation*, *Web Copy: http://www. glue. umd. edu/ajaleel/workload* (2010).

[14] A. R. Alameldeen and D. A. Wood, *Addressing workload variability in architectural simulations*, *IEEE Micro* **23** (Nov, 2003) 94–98.

[15] A. R. Alameldeen and D. A. Wood, *Variability in architectural simulations of multi-threaded workloads*, in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pp. 7–18, Feb, 2003.

[16] M. Guevara, B. Lubin, and B. C. Lee, *Strategies for anticipating risk in heterogeneous system design*, in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 154–164, Feb, 2014.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, *The gem5 simulator*, *SIGARCH Comput. Archit. News* **39** (Aug., 2011) 1–7.

[18] M. D. Hill and M. R. Marty, *Amdahl's law in the multicore era*, *Computer* **41** (July, 2008) 33–38.

[19] W. Cui and T. Sherwood, *Estimating and understanding architectural risk*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, 2017.

[20] W. Cui, Y. Ding, D. Dangwal, A. Holmes, J. McMahan, A. Javadi-Abhari, G. Tzimpragos, F. Chong, and T. Sherwood, *Charm: A language for closed-form high-level architecture modeling*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 152–165, June, 2018.

[21] Z. He, W. Cui, C. Cui, T. Sherwood, and Z. Zhang, *Efficient uncertainty modeling for system design via mixed integer programming*, in *Proc. Intl. Conf. Computer-Aided Design, Nov*, pp. 0278–0070, 2019.

[22] S. Kaplan and B. J. Garrick, *On the quantitative definition of risk*, *Risk Analysis* **1** (1981), no. 1 11–27.

[23] B. C. Lee and D. M. Brooks, *Accurate and efficient regression modeling for microarchitectural performance and power prediction*, in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 185–194, ACM, 2006.

[24] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, *Discovering and exploiting program phases*, *IEEE Micro* **23** (Nov, 2003) 84–93.

[25] C. Dubach, T. Jones, and M. O'Boyle, *Microarchitectural design space exploration using an architecture-centric approach*, in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 262–271, IEEE Computer Society, 2007.

[26] M. S. B. Altaf and D. A. Wood, *Logca: A high-level performance model for hardware accelerators*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 375–388, ACM, 2017.

[27] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, *Dark silicon and the end of multicore scaling*, in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[28] M. Hill and V. Janapa Reddi, *Gables: A roofline model for mobile socs*, in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 317–330, Feb, 2019.

[29] D. H. Woo, D. H. Woo, D. H. Woo, D. H. Woo, H. H. S. Lee, H. H. S. Lee, H. H. S. Lee, and H. H. S. Lee, *Extending amdahl's law for energy-efficient computing in the many-core era*, *Computer* **41** (Dec, 2008) 24–31.

[30] X.-H. Sun and Y. Chen, *Reevaluating amdahl's law in the multicore era*, *Journal of Parallel and Distributed Computing* **70** (2010), no. 2 183 – 188.

[31] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, *Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?*, in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2010.

[32] J. L. Gustafson, *Reevaluating amdahl's law*, *Commun. ACM* **31** (May, 1988) 532–533.

[33] D. Brooks, V. Tiwari, and M. Martonosi, *Wattch: A framework for architectural-level power analysis and optimizations*, in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 83–94, ACM, 2000.

[34] A. Hartstein and T. R. Puzak, *The optimum pipeline depth for a microprocessor*, in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, (Washington, DC, USA), pp. 7–13, IEEE Computer Society, 2002.

[35] B. Lee and D. Brooks, *Statistically rigorous regression modeling for the microprocessor design space*, in *ISCA-33: Workshop on Modeling, Benchmarking, and Simulation*, 2006.

[36] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*, in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 195–206, ACM, 2006.

[37] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, *Methods of inference and learning for performance modeling of parallel applications*, in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, (New York, NY, USA), pp. 249–258, ACM, 2007.

[38] B. C. Lee and D. M. Brooks, *Illustrative design space studies with microarchitectural regression models*, in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 340–351, Feb, 2007.

[39] B. C. Lee, J. Collins, H. Wang, and D. Brooks, *Cpr: Composable performance regression for scalable multiprocessor models*, in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 270–281, IEEE Computer Society, 2008.

[40] B. C. Lee and D. Brooks, *Efficiency trends and limits from comprehensive microarchitectural adaptivity*, in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 36–47, ACM, 2008.

[41] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, *Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures*, in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–480, Dec, 2009.

[42] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, *Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis*, in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 26–36, ACM, 2010.

[43] M. Oskin, F. T. Chong, and M. Farrens, *Hls: Combining statistical and symbolic simulation to guide microprocessor designs*, in *Proceedings of the 27th Annual*

International Symposium on Computer Architecture, ISCA '00, (New York, NY, USA), pp. 71–82, ACM, 2000.

[44] J. Bornholt, T. Mytkowicz, and K. S. McKinley, *Uncertain¡t¿: A first-order type for uncertain data*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 51–66, ACM, 2014.

[45] M. Guevara, B. Lubin, and B. C. Lee, *Navigating heterogeneous processors with market mechanisms*, in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 95–106, Feb, 2013.

[46] S. M. Zahedi and B. C. Lee, *Ref: Resource elasticity fairness with sharing incentives for multiprocessors*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 145–160, ACM, 2014.

[47] S. Fan, S. M. Zahedi, and B. C. Lee, *The computational sprinting game*, in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 561–575, ACM, 2016.

[48] S. Borkar, *Thousand core chips: A technology perspective*, in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, (New York, NY, USA), pp. 746–749, ACM, 2007.

[49] L. Yavits, A. Morad, and R. Ginosar, *The effect of communication and synchronization on amdahl's law in multicore systems*, *Parallel Computing* **40** (2014), no. 1 1 – 16.

[50] J. A. Cunningham, *The use and evaluation of yield models in integrated circuit manufacturing*, *IEEE Transactions on Semiconductor Manufacturing* **3** (May, 1990) 60–71.

[51] K. Constantinides, O. Mutlu, and T. Austin, *Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation*, in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 282–293, Nov, 2008.

[52] H. D. Foster, *Trends in functional verification: A 2014 industry study*, in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June, 2015.

[53] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, *Varius: A model of process variation and resulting timing errors for microarchitects*, *IEEE Transactions on Semiconductor Manufacturing* **21** (Feb, 2008) 3–13.

[54] X. Liang and D. Brooks, *Mitigating the impact of process variations on processor register files and execution units*, in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 504–514, Dec, 2006.

[55] A. Rahimi, L. Benini, and R. K. Gupta, *Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software*, *Proceedings of the IEEE* **104** (July, 2016) 1410–1448.

[56] M. Bhadauria, V. M. Weaver, and S. A. McKee, *Understanding parsec performance on contemporary cmps*, in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 98–107, Oct, 2009.

[57] G. E. P. Box and D. R. Cox, *An analysis of transformations*, *Journal of the Royal Statistical Society. Series B (Methodological)* **26** (1964), no. 2 211–252.

[58] D. W. Scott, *Kernel Density Estimators*, pp. 125–193. John Wiley Sons, Inc., 2008.

[59] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016.

[60] A. Lee, *Real-time latin-hypercube-sampling-based Monte Carlo Error Propagation*, 2014.

[61] I. Koren and Z. Koren, *Defect tolerance in vlsi circuits: techniques and yield analysis*, *Proceedings of the IEEE* **86** (Sep, 1998) 1819–1838.

[62] Intel, "Intel processor pricing." `https://www.intc.com/investor-relations/investor-education-and-news/cpu-price-list`, 2017. Accessed: 2017-04-03.

[63] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, *Dark silicon and the end of multicore scaling*, in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[64] M. D. Hill and M. R. Marty, *Amdahl's law in the multicore era*, *Computer* **41** (July, 2008) 33–38.

[65] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, *Many-core vs. many-thread machines: Stay away from the valley*, *IEEE Computer Architecture Letters* **8** (Jan, 2009) 25–28.

[66] S. Borkar, *The exascale challenge*, in *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, pp. 2–3, April, 2010.

[67] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen, *Validating the unit correctness of spreadsheet programs*, in *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, (Washington, DC, USA), pp. 439–448, IEEE Computer Society, 2004.

[68] S. G. Powell, K. R. Baker, and B. Lawson, *A critical review of the literature on spreadsheet errors*, *Decis. Support Syst.* **46** (Dec., 2008) 128–138.

[69] L. De Moura and N. Bjørner, *Z3: An efficient smt solver*, in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[70] H. Elmqvist, S. Mattsson, H. Elmqvist, and D. Ab, *An introduction to the physical modeling language modelica*, in *Proc. 9th European Simulation Sympossium ESS97, SCS Int*, pp. 110–114, 1997.

[71] K. L. Spafford and J. S. Vetter, *Aspen: A domain specific language for performance modeling*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 84:1–84:11, IEEE Computer Society Press, 2012.

[72] N. R. Tallent and A. Hoisie, *Palm: Easing the burden of analytical performance modeling*, in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), pp. 221–230, ACM, 2014.

[73] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, *The liberty simulation environment: A deliberate approach to high-level system modeling*, *ACM Trans. Comput. Syst.* **24** (Aug., 2006) 211–249.

[74] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, *Exasat: An exascale co-design tool for performance modeling*, *Int. J. High Perform. Comput. Appl.* **29** (May, 2015) 209–232.

[75] S. R. Alam and J. S. Vetter, *A framework to develop symbolic performance models of parallel applications*, in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, (Washington, DC, USA), pp. 320–320, IEEE Computer Society, 2006.

[76] S. R. Alam and J. S. Vetter, *Hierarchical model validation of symbolic performance models of scientific kernels*, in *European Conference on Parallel Processing*, pp. 65–77, Springer, 2006.

[77] K. H. Temme, *Charm: a synthesis tool for high-level chip-architecture planning*, in *1989 Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 4.2/1–4.2/4, May, 1989.

[78] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous data flow programming language lustre*, *Proceedings of the IEEE* **79** (Sep, 1991) 1305–1320.

[79] L. Mandel and M. Pouzet, *Reactiveml: A reactive extension to ml*, in *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, (New York, NY, USA), pp. 82–93, ACM, 2005.

[80] M. A. Hammer, U. A. Acar, and Y. Chen, *Ceal: A c-based language for self-adjusting computation*, in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 25–37, ACM, 2009.

[81] T. Szabó, S. Erdweg, and M. Voelter, *Inca: A dsl for the definition of incremental program analyses*, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 320–331, ACM, 2016.

[82] P. LeGuernic, T. Gautier, M. L. Borgne, and C. L. Maire, *Programming real-time applications with signal*, *Proceedings of the IEEE* **79** (Sep, 1991) 1321–1336.

[83] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, *Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation*, in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, (New York, NY, USA), pp. 18–27, ACM, 2002.

[84] P. Manolios, D. Vroon, and G. Subramanian, *Automating component-based system assembly*, in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, (New York, NY, USA), pp. 61–72, ACM, 2007.

[85] C. Haubelt, J. Teich, R. Feldmann, and B. Monien, *Sat-based techniques in system synthesis*, in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, (Washington, DC, USA), pp. 11168–, IEEE Computer Society, 2003.

[86] S. Peter and T. Givargis, *Component-based synthesis of embedded systems using satisfiability modulo theories*, *ACM Trans. Des. Autom. Electron. Syst.* **20** (Sept., 2015) 49:1–49:27.

[87] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich, *Improving platform-based system synthesis by satisfiability modulo theories solving*, in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, (New York, NY, USA), pp. 135–144, ACM, 2010.

[88] W. Liu, Z. Gu, J. Xu, X. Wu, and Y. Ye, *Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems*, *IEEE Transactions on Parallel and Distributed Systems* **22** (Aug, 2011) 1382–1389.

[89] T. E. Sheard, *Painless programming combining reduction and search: Design principles for embedding decision procedures in high-level languages*, in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, (New York, NY, USA), pp. 89–102, ACM, 2012.

[90] W. R. Inc., "Mathematica, Version 11.2." Champaign, IL, 2017.

[91] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, *Optimizing fpga-based accelerator design for deep convolutional neural networks*, in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[92] S. Bravyi and J. Haah, *Magic-state distillation with low overhead*, *Physical Review A* **86** (2012), no. 5 052329.

[93] A. G. Fowler, S. J. Devitt, and C. Jones, *Surface code implementation of block code state distillation*, *Scientific Reports* **3** (jun, 2013) 1939.

[94] J. O'Gorman and E. T. Campbell, *Quantum computation with realistic magic-state factories*, *Physical Review A* **95** (2017), no. 3 032338.

[95] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, *Commun. ACM* **52** (Apr., 2009) 65–76.

[96] M. Courbariaux, Y. Bengio, and J.-P. David, *Binaryconnect: Training deep neural networks with binary weights during propagations*, in *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

[97] J. Alwen and J. Blocki, *Efficiently computing data-independent memory-hard functions*, in *Annual Cryptology Conference*, pp. 241–271, Springer, 2016.

[98] B. C. Lee and D. M. Brooks, *Accurate and efficient regression modeling for microarchitectural performance and power prediction*, in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 185–194, ACM, 2006.

[99] D. Xiu and G. E. Karniadakis, *The wiener–askey polynomial chaos for stochastic differential equations*, *SIAM journal on scientific computing* **24** (2002), no. 2 619–644.

[100] D. Xiu and G. E. Karniadakis, *Modeling uncertainty in flow simulations via generalized polynomial chaos*, *Journal of computational physics* **187** (2003), no. 1 137–167.

[101] Q. Guo, T. Chen, Y. Chen, and F. Franchetti, *Accelerating architectural simulation via statistical techniques: A survey*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35** (March, 2016) 433–446.

[102] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*, in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 195–206, ACM, 2006.

[103] P. J. Joseph, , and M. J. Thazhuthaveetil, *Construction and use of linear regression models for processor performance analysis*, in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pp. 99–108, Feb, 2006.

[104] J. Chen, M. D. Flood, and R. B. Sowers, *Measuring the unmeasurable: an application of uncertainty quantification to treasury bond portfolios*, *Quantitative Finance* **17** (2017), no. 10 1491–1507, [https://doi.org/10.1080/14697688.2017.1296176].

[105] Y. Qian, C. Jackson, F. Giorgi, B. Booth, Q. Duan, C. Forest, D. Higdon, Z. J. Hou, and G. Huerta, *Uncertainty quantification in climate modeling and projection*, *Bulletin of the American Meteorological Society* **97** (2016), no. 5 821–824, [https://doi.org/10.1175/BAMS-D-15-00297.1].

[106] S. Marino, I. B. Hogue, C. J. Ray, and D. E. Kirschner, *A methodology for performing global uncertainty and sensitivity analysis in systems biology*, *Journal of Theoretical Biology* **254** (2008), no. 1 178 – 196.

[107] H. N. Najm, *Uncertainty quantification and polynomial chaos techniques in computational fluid dynamics*, *Annual Review of Fluid Mechanics* **41** (2009), no. 1 35–52, [https://doi.org/10.1146/annurev.fluid.010908.165248].

[108] Z. Zhang, X. Yang, I. V. Oseledets, G. E. Karniadakis, and L. Daniel, *Enabling high-dimensional hierarchical uncertainty quantification by anova and tensor-train decomposition*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34** (Jan, 2015) 63–76.

[109] S. Bhardwaj and S. B. K. Vrudhula, *Leakage minimization of nano-scale circuits in the presence of systematic and random variations*, in *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, (New York, NY, USA), pp. 541–546, ACM, 2005.

[110] L. Zhang, L. S. Bai, R. P. Dick, L. Shang, and R. Joseph, *Process variation characterization of chip-level multiprocessors*, in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 694–697, ACM, 2009.

[111] H.-Y. Wong, L. Cheng, Y. Lin, and L. He, *Fpga device and architecture evaluation considering process variations*, in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, (Washington, DC, USA), pp. 19–24, IEEE Computer Society, 2005.

[112] A. Das, S. Ozdemir, G. Memik, J. Zambreno, and A. Choudhary, *Mitigating the effects of process variations: Architectural approaches for improving batch performance*, in *Workshop on Architectural Support for Gigascale Integration (ASGI), San Diego, CA*, 2007.

[113] G. Yan, F. Sun, H. Li, and X. Li, *Corerank: Redeeming amp;x201c;sick silicon amp;x201d; by dynamically quantifying core-level healthy condition*, *IEEE Transactions on Computers* **65** (March, 2016) 716–729.

[114] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou, *Yield-aware cache architectures*, in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 15–25, Dec, 2006.

[115] X. Liang, R. Canal, G. Wei, and D. Brooks, *Process variation tolerant 3t1d-based cache architectures*, in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 15–26, Dec, 2007.

[116] R. L. Iman and J. C. Helton, *An investigation of uncertainty and sensitivity analysis techniques for computer models*, *Risk Analysis* **8** no. 1 71–90, [https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1539-6924.1988.tb01155.x].

[117] A. Saltelli, *Sensitivity analysis for importance assessment*, *Risk Analysis* **22** no. 3 579–590, [https://onlinelibrary.wiley.com/doi/pdf/10.1111/0272-4332.00040].

[118] Q. C. Curran, D. Allaire, and K. E. Willcox, *Sensitivity analysis methods for mitigating uncertainty in engineering system design*, *Systems Engineering* **21** no. 3 191–209, [https://onlinelibrary.wiley.com/doi/pdf/10.1002/sys.21422].

[119] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria, *A sensitivity-based design space exploration methodology for embedded systems*, *Design Automation for Embedded Systems* **7** (Sep, 2002) 7–33.

[120] K. De Vogeleer, G. Memmi, and P. Jouvelot, *Parameter sensitivity analysis of the energy/frequency convexity rule for application processors*, *Sustainable Computing: Informatics and Systems* **15** (2017) 16–27.

[121] Y. Zhu and W. Wong, *Sensitivity analysis of a superscalar processor model*, in *Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002)* (F. Lai and J. Morris, eds.), vol. 6 of *CRPIT*, (Melbourne, Australia), pp. 109–118, ACS, 2002.

[122] M. Arora, F. Wang, B. Rychlik, and D. M. Tullsen, *Efficient system design using the statistical analysis of architectural bottlenecks methodology*, in *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pp. 217–226, IEEE, 2012.

[123] B. Liu, Q. Zhang, and G. G. Gielen, *A gaussian process surrogate model assisted evolutionary algorithm for medium scale expensive optimization problems*, *IEEE Transactions on Evolutionary Computation* **18** (2014), no. 2 180–192.

[124] J. Eason and S. Cremaschi, *Adaptive sequential sampling for surrogate model generation with artificial neural networks*, *Computers & Chemical Engineering* **68** (2014) 220–232.

[125] F. Girosi, *An equivalence between sparse approximation and support vector machines*, *Neural computation* **10** (1998), no. 6 1455–1480.

[126] X. Li, J. Le, P. Gopalakrishnan, and L. T. Pileggi, *Asymptotic probability extraction for nonnormal performance distributions*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **26** (2007), no. 1 16–37.

[127] M. Frangos, Y. Marzouk, K. Willcox, and B. van Bloemen Waanders, *Surrogate and reduced-order modeling: A comparison of approaches for large-scale statistical inverse problems*, *Large-Scale Inverse Problems and Quantification of Uncertainty* 123–149.

[128] D. Xiu, *Numerical methods for stochastic computations: a spectral method approach*. Princeton university press, 2010.

[129] Z. Zhang, I. A. M. Elfadel, and L. Daniel, *Uncertainty quantification for integrated circuits: Stochastic spectral methods*, in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pp. 803–810, IEEE, 2013.

[130] Z. Zhang, T. A. El-Moselhy, I. M. Elfadel, and L. Daniel, *Calculation of generalized polynomial-chaos basis functions and gauss quadrature rules in hierarchical uncertainty quantification*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **33** (2014), no. 5 728–740.

[131] Z. Zhang, T.-W. Weng, and L. Daniel, *Big-data tensor recovery for high-dimensional uncertainty quantification of process variations*, *IEEE Transactions on Components, Packaging and Manufacturing Technology* **7** (2017), no. 5 687–697.

[132] C. Cui and Z. Zhang, *Stochastic collocation with non-gaussian correlated process variations: Theory, algorithms and applications*, *IEEE Transactions on Components, Packaging and Manufacturing Technology* (2018).

[133] Q. Guo, T. Chen, Y. Chen, and F. Franchetti, *Accelerating architectural simulation via statistical techniques: A survey*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35** (2016), no. 3 433–446.

[134] Z. Zhang, T. A. El-Moselhy, I. M. Elfadel, and L. Daniel, *Stochastic testing method for transistor-level uncertainty quantification based on generalized polynomial chaos*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **32** (2013), no. 10 1533–1545.

[135] I. M. Sobol, *Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates*, *Mathematics and computers in simulation* **55** (2001), no. 1-3 271–280.

[136] J. C. Helton and F. J. Davis, *Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems*, *Reliability Engineering & System Safety* **81** (2003), no. 1 23–69.

[137] D. Xiu, *Fast numerical methods for stochastic computations: a review*, *Communications in computational physics* **5** (2009), no. 2-4 242–272.

[138] C. Cui and Z. Zhang, *Uncertainty quantification of electronic and photonic ics with non-gaussian correlated process variations*, in *Proceedings of the International Conference on Computer-Aided Design*, p. 97, ACM, 2018.

[139] H. Wu, Y. Zhou, S. Dong, and Y. Song, *Probabilistic load flow based on generalized polynomial chaos*, *IEEE Trans. Power Syst* **32** (2017), no. 1 820–821.

[140] G. H. Golub and J. H. Welsch, *Calculation of gauss quadrature rules*, *Mathematics of computation* **23** (1969), no. 106 221–230.

[141] D. Xiu and J. S. Hesthaven, *High-order collocation methods for differential equations with random inputs*, *SIAM Journal on Scientific Computing* **27** (2005), no. 3 1118–1139.

[142] M. M. Sabry Aly, M. Gao, G. Hills, C. Lee, G. Pitner, M. M. Shulaker, T. F. Wu, M. Asheghi, J. Bokor, F. Franchetti, K. E. Goodson, C. Kozyrakis, I. Markov, K. Olukotun, L. Pileggi, E. Pop, J. Rabaey, C. R, H. . P. Wong, and S. Mitra, *Energy-efficient abundant-data computing: The n3xt 1,000x*, *Computer* **48** (Dec, 2015) 24–33.

[143] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, *Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques*, in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, (Piscataway, NJ, USA), pp. 694–701, IEEE Press, 2011.

[144] B. Zhai, D. Blaauw, D. Sylvester, D. Sylvester, and K. Flautner, *Theoretical and practical limits of dynamic voltage scaling*, in *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, (New York, NY, USA), pp. 868–873, ACM, 2004.

[145] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January, 2011.

[146] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, *Rodinia: A benchmark suite for heterogeneous computing*, in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct, 2009.

[147] T. N. Miller, R. Thomas, and R. Teodorescu, *Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units*, *IEEE Computer Architecture Letters* **11** (July, 2012) 45–48.

[148] I. M. Sobolá, *Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates*, *Math. Comput. Simul.* **55** (Feb., 2001) 271–280.

[149] J. Bornholt, T. Mytkowicz, and K. S. McKinley, *Uncertain<t>: A first-order type for uncertain data*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 51–66, ACM, 2014.

[150] J. McMahan, W. Cui, L. Xia, J. Heckey, F. T. Chong, and T. Sherwood, *Challenging on-chip sram security with boot-state statistics*, in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 101–105, May, 2017.