Bespoke Processors for Embedded Systems and Secure Multi-Party Computation

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Shashank Hegde

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John M Sartori

May, 2023

© Shashank Hegde 2023
ALL RIGHTS RESERVED

Acknowledgements

With all of my heart, I want to express my deepest gratitude to my academic advisor, Prof. John Sartori. He has been an unwavering beacon of guidance and support throughout my PhD journey. Without his constant presence and immense wisdom, I would have been lost in a sea of uncertainty and doubt.

Prof. Sartori has been more than just an advisor to me - he has been a mentor, a friend, and a confidant. He has always made himself available, day or night, for any help or advice that I needed, both professionally and personally. His unwavering dedication to my success has been a source of inspiration and motivation that has propelled me forward in the most difficult times.

What truly sets Prof. Sartori apart is his unshakable belief in me and my abilities. He has always encouraged me to pursue any research idea that interested me without any restrictions, giving me the freedom to explore and innovate in ways that I never thought possible. His trust in my potential has instilled a sense of confidence and determination that I will carry with me for the rest of my life.

I also want to extend my heartfelt appreciation to the National Science Foundation for their invaluable support of my research through the NSF grants. I would also like to thank my other collaborators, Hari Cherupalli, Henry Duwe, Zhenhuan Zhang, Tal Davidi, and Ruiyu Zhu without whom, my research work would not be complete. I further like to express my sincere thanks to the technical staff at the University, Chimai Nguyen and Carlos Soria for, their assistance and support.

My heart is overflowing with gratitude as I reflect on my PhD journey and the people who made it possible. I owe an immense debt of gratitude to my labmate and collaborator, Hari Cherupalli. Hari has been a constant source of guidance and counsel throughout my research endeavors, generously spending countless hours working

alongside me on my projects, even after he had graduated from the University. His dedication to our shared goals has been a guiding light that has kept me motivated and focused through the most challenging times.

I would also like to extend a special thanks to Subhash Sethumurugan, whose partnership in my PhD has been nothing short of transformative. Over the years, Subhash has become more than just a colleague - he is a dear friend and mentor, whose influence has left an indelible mark on my life and work. Through our many discussions and debates, I have learned so much from him, not only about research but also about life and the importance of staying curious and open-minded.

Of course, my PhD journey would not have been complete without the support of all my other labmates - Himanshoo Sahoo, Shaman Narayanan, Nishanth Somashekara Murthy, and Tariq Azmy. These wonderful individuals have been a constant source of inspiration and encouragement, and their support has helped me to persevere and achieve my academic goals.

I consider myself fortunate to have had the privilege of working with esteemed scholars and intellectuals and I am certainly happy to have them in my exam committee. The expertise and guidance provided by Prof. Kia Bazargan, Prof. Marc Riedel, and Prof. Jon Weissman have undoubtedly played a significant role in shaping the outcome of my research. I extend my heartfelt gratitude to each of them for their time, wisdom, and unwavering commitment to my academic success.

Abstract

With transistor scaling nearing atomic dimensions and leakage power dissipation imposing strict energy limitations, it has become increasingly difficult to improve energy efficiency in modern processors without sacrificing performance and functionality. One way to avoid this tradeoff and reduce energy without reducing performance or functionality is to take a cue from application behavior and eliminate energy in areas that will not impact application performance. This approach is especially relevant in embedded systems, which often have ultra-low power and energy requirements and typically run a single application over and over throughout their operational lifetime. In such processors, application behavior can be effectively characterized and leveraged to identify opportunities for "free" energy savings. We find that in addition to instruction-level sequencing, constraints imposed by program-level semantics can be used to automate processor customization and further improve energy efficiency. This dissertation describes automated techniques to identify, form, propagate, and enforce application-based constraints in gate-level simulation to reveal opportunities to optimize a processor at the design level. While this can significantly improve energy efficiency, if the goal is truly to maximize energy efficiency, it is important to consider not only design-level optimizations but also architectural optimizations. That being said, architectural optimization presents several challenges. First, the symbolic simulation tool used to characterize gate-level behavior of an application must be written anew for each new architecture. Given the expansiveness of the architectural parameter space, this is not feasible. To overcome this barrier, we developed a generic symbolic simulation tool that can handle any design, technology, or architecture, making it possible to explore application-specific architectural optimizations. However, exploring each parameter variation still requires synthesizing a new design and performing application-specific optimizations, which again becomes infeasible due to the large architecture parameter space. Given the wide usage of Machine Learning (ML) for effective design space exploration, we sought the aid of ML to efficiently explore the architectural parameter space. We built a tool that takes into account the impacts of architectural optimizations on an application and predicts the architectural parameters that result in near-optimal energy

efficiency for an application. This dissertation explores the objective, training, and inference of the ML model in detail.

Bespoke processors are tailored for a particular application and provide significantly greater energy efficiency than a general-purpose architecture executing that application. Given the paramount importance of data, privacy, and security in today's data-driven landscape, we have tailored a bespoke domain-specific processor for Secure Multi-Party Computation (MPC). The MPC computing paradigm is fundamentally different than traditional general-purpose computing; it allows multiple parties to perform collaborative computations on shared data without revealing any of the private data that they own, demonstrating benefits in several application domains including machine learning, data analytics, and privacy preservation. Our bespoke MPC architecture encompasses a complete end-to-end solution, including the compiler and assembler, a new ISA, and the processor architecture. Our bespoke processor architecture addresses the bottlenecks of existing MPC systems and improves efficiency significantly, enabling the use of MPC in new applications where overheads were previously prohibitive.

Contents

Acknowledgements **i**

Abstract **iii**

Contents **v**

List of Tables **viii**

List of Figures **x**

1 Introduction **1**

2 Background **7**

2.1 Symbolic Simulation Based Hardware Software Co-analysis 7

2.2 Conservative State 8

3 Constrained Conservative Symbolic Hardware-Software Co-analysis **10**

3.1 Conservative State Limitation 10

3.2 Proposed Work 15

3.2.1 Encoding Constraints From Binary 16

3.2.2 Propagating Constraints 18

3.2.3 Enforcing Constraints 19

3.3 Proof of CCS Correctness 20

3.4 Evaluation 21

3.4.1 Analysis Time 23

3.4.2 Exercisable Gates 24

3.4.3 Final Remarks 26
3.5 Related Work 27
3.5.1 Static Analysis 27
3.5.2 Hardware-Software Co-analysis 27
3.6 Summary 28

4 Design-Agnostic Symbolic Co-analysis Tool 29
4.1 Gate-Level Simulator 29
4.2 Extending Iverilog For Symbolic Hardware-Software Co-Analysis 30
4.2.1 Iverilog Software Flow Enhancement 31
4.3 Symbolic Hardware-software Co-analysis Using Iverilog 32
4.3.1 Designing A Testbench For Symbolic Hardware-Software Co-Analysis For Iverilog 33
4.3.2 Conservative State Management 35
4.3.3 Propagation Of Symbols 37
4.4 Evaluation 38
4.4.1 Validation 39
4.4.2 Exercisable Gates 40
4.4.3 Simulation paths 40
4.5 Related Work 43
4.6 Summary 44

5 Application-Specific Architecture Selection 46
5.1 Effect Of Bespoke Process On Architectural Variants 47
5.2 Effect Of Architectural Variants On Efficiency Metric 49
5.2.1 Processor Architectures 49
5.2.2 Hardware Accelerators 51
5.3 Motivation 51
5.4 Application-Specific Architecture Selection 53
5.4.1 Feature Extraction 55
5.4.2 Model Selection 56
5.4.3 Training The Model 57
5.4.4 Prediction, Ranking, And Architecture Selection 57

5.4.5 Application-Specific Architecture Selection For DSP Circuits . . 58
5.5 Evaluation 58
5.5.1 Design Space Exploration For Bespoke General Purpose Processors 60
5.5.2 Design Space Exploration For Bespoke DSP Accelerators 66
5.5.3 Final Remarks 67
5.6 Generality And Limitations 70
5.7 Related Work 71
5.7.1 Design Space Exploration 71
5.7.2 Application-Specific Processor Cores And High-Level Synthesis . 72
5.8 Summary 72

6 Bespoke Domain-specific Architecture for Secure Multi-Party Computation 74
6.1 Introduction to Multi-partly Computation 75
6.2 Background on MPC 77
6.2.1 Logic implementation using XOR-Secret Share(XOR-SS) 78
6.3 Related Work 83
6.4 Bespoke Processors for Secure MPC 85
6.4.1 Compiler and Assembler design 85
6.4.2 MPC ISA 88
6.4.3 Hardware Architecture 93
6.4.4 MPC processor description 99
6.5 Methodology 99
6.5.1 Baseline setup 100
6.5.2 BENCHMARKS 100
6.6 Results 100
6.6.1 Performance variation with network speed 101
6.6.2 Comparison with baseline 104
6.7 Conclusion 108

7 Conclusion And Discussion 109

References 112

List of Tables

3.1 Benchmarks 21

3.2 Constrained conservative state symbolic co-analysis reduces analysis time compared to naive and conservative state-based co-analysis and enables analysis of applications with complex control structures. 22

3.3 Use of constraints reduces the number of explored symbolic execution paths. 22

3.4 Use of constraints reduces the number of gates identified as exercisable. 22

3.5 Symbolic simulation approach comparison. 26

4.1 Benchmark applications 39

4.2 Target platform characterization 39

4.3 Gate count analysis 40

4.4 Simulation path and runtime analysis 42

5.1 General purpose processors evaluated 59

5.2 DSP accelerators evaluated 60

5.3 Architectural variants explored 61

5.4 Benchmark applications for general purpose processors 62

5.5 Summary of optimal architecture in terms of area; the model predicts with 100% accuracy in top 10 predictions 64

5.6 Summary of optimal architecture in terms of **energy per bit**; the model predicts with 100% accuracy in top 10% of predictions 66

5.7 This table presents the optimal architectural variant for each bespoke accelerator and its rank as predicted by our model for **area**. In most cases, our model ranks the optimal architecture within the top 10% of candidate architectures. 68

5.8 This table presents the optimal architectural variant for each bespoke accelerator and its rank as predicted by our model for **energy**. In most cases, our model ranks the optimal architecture within the top 10% of candidate architectures. 69
6.1 Analysis of AND gate complexity with different Adders 98
6.2 Baseline configuration 100
6.3 Benchmark Programs 101
6.4 Benchmark results shows the runtime of various benchmarks to compute and final reveal. Reveal here refers to decrypting the final result by sharing the result shares and performing the XOR operation on the result. 104

List of Figures

2.1 Example of conservative state representing register values at different execution states for the same PC. 9
3.1 Example C program. 11
3.2 Compiled MSP430 program. 11
3.3 Conservative state-based scalable symbolic co-analysis can analyze applications with infinite loops and input-dependent branches by simulating conservative states that capture the activity of multiple possible states. 13
3.4 Constraining memory elements based on bounds from the software level reduces pessimism in estimating the number of gates marked as exercisable and also reduces the number of paths that need to be explored. . . 14
3.5 Methodology for CCS 15
3.6 Example of constraint encoding during static analysis of the application binary. 17
4.1 We add a new type of event to capture 'symbolic events' in iverilog's event queue. This enables us to monitor control signals for X and halt the simulation when necessary. The VVP engine is a part of iverilog source that executes an iverilog compiled assembly code that is generated from the verilog testbench. 31
4.2 Our design-agnostic symbolic co-analysis tool is built on top of iverilog to allow hardware-software co-analysis of any digital design. 32

4.3 Various approaches for conservative state generation exhibit trade-offs between simulation effort and conservative over-approximation. To capture all states in the first row (green) we could either create two conservative states as shown in the second row (blue) or one uber-conservative state as shown in the third row (red). 36
4.4 Our symbolic tool allows rules for symbol propagation to be customized. The left sub-figure shows a case where circuit inputs are propagated as separate symbolic values, while the right sub-figure shows a case where the symbolic values carry no identifying information and thus cannot be distinguished. 37
4.5 Benchmarks run on MSP430 processor have a higher reduction in exercisable gate count compared to MIPS and RISCV processors because of the presence of unused peripherals in MSP430. 41
4.6 Benchmarks run on MIPS and RISCV processors have a higher number of simulated paths because a 16-bit register is used to indicate branch conditions, whereas in MSP430, a 1-bit register is used, resulting in fewer conservative states. 43
5.1 This plot shows the total gate count for various architectural variants of the darkriscv processor both before and after bespoke customization for the tea8 application. The optimal processor variant before customization (green diamond) is different than the optimal bespoke processor variant after customization (green star). 48
5.2 The plots compare per-bit energy consumption (top) and area (bottom) for bespoke processors tailored for mult and binsearch applications, starting from three distinct processor architectures – MSP430, MIPS, and RISC-V. The MSP430-based bespoke processor has the lowest per-bit energy consumption for the mult application, but the MIPS-based design has the lowest energy for the binsearch application. On the other hand, the RISC-V-based design has the lowest per-bit area for each application. The results demonstrate that the best processor architecture from which to generate a bespoke processor differs based on the target application and efficiency metric. 50

5.3 The plots compare energy consumption (top) and area (bottom) for bespoke accelerators tailored for applications with different input bit precision, starting from folded and un-folded architectural variants of a 32-bit DCT filter DSP accelerator. The x-axis represents input signal bit width, corresponding to different applications that require different levels of precision. Bespoke accelerators generated from the folded architecture have lower area for all input bit widths, but for a bit width of 32, the unfolded accelerator has lower energy due to its lower computation time. The best accelerator architecture from which to generate a bespoke accelerator differs based on the target application (bit width) and efficiency metric. 52
5.4 Our machine learning model for selecting an architectural configuration from which to generate a bespoke processor uses architecture features of the baseline design and application characteristics to predict metric values for each architectural configuration in the architectural parameter space. A short-list of candidate architectures is evaluated more thoroughly to identify the most efficient architectural variant. 54
5.5 We use a neural network that predicts different desired metrics. There is a slight variation in the models that predict area and energy metrics. The model that predicts energy uses the tanh activation function, while the model that predicts area uses ReLU activation. 57
5.6 This plot shows the normalized energy per bit (top) and normalized area per bit (bottom) predictions of bespoke processors for mult. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend. 63
5.7 This plot shows predicted and actual values for normalized energy per bit (top) and normalized area per bit (bottom) for bespoke DCT accelerators. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend, indicating that our model can be used to predict the optimal architecture. 65

6.1 The XOR function can be computed without interleaving communication with the computation. In this example, Alice and Bob own 0xA4 and 0x2B, respectively. XOR is computed by generating secret shares, exchanging the shares, and having each party compute XOR independently. Finally, the result is revealed by opening the shares. 81
6.2 High-level flow for compiling a front-end application in Python into an MPC-friendly representation enabling the assembler to generate optimal machine code. 88
6.3 Encoding and boundary description for different classes of instructions that allows vectored instruction with efficient packing. 90
6.4 Conditional branching statements are grouped as ternary operations based on the condition; the assembler implements $>, <, >=, <=$ as a part of a subtract operation where the N, Z, C, V flags can be reused to evaluate different conditional outcomes without having to re-evaluate the conditional outcome for every different condition involving the same result. This reduces the amount of communication required. 93
6.5 Architecture diagram of the execution unit. The scheduled instruction that needs communication is put on the conduit in order and transmitted to the other party. Upon receiving the partial terms from the other party, corresponding operations continue their operation. Finally, upon completion, the status bits are updated in the scoreboard. 94
6.6 The performance of MPC computation is directly correlated to network speed. As the network speed gets closer to the frequency of our hardware, network no longer becomes the bottleneck. This plot is for the processor running at 400MHz. 102
6.7 Resource utilization: shows the total number of AND, XOR, and Composite AND operations that were scheduled for different benchmarks. . 103
6.8 Shows the variation of runtime in microseconds for various benchmarks with the baseline results from the PCF platform [1] consisting of two schedule modes – Eager and Lazy. 106

6.9 Shows the network usage and the number of tuples used for various benchmarks. Tuples used and network traffic mainly depend on the number of *AND* and *compositeAND* operations. Except for *sum*, the rest of the benchmarks consists of conditional instructions which use the *composite-AND* gate; thus, the tx-traffic is reduced. 107

Chapter 1

Introduction

One of the main challenges that processor architects face in recent times is improving the energy efficiency of a design. Dennard Scaling has ended and transistor size scaling has slowed down. Irrespective of scale, from servers to embedded systems, improving energy efficiency is becoming increasingly challenging. Energy efficiecy is especially vital for embedded systems that run applications such as implantables [2, 3], wearables [4, 5], and IoT applications [6–11], since these systems are often powered by batteries or energy harvesting. One defining characteristic of such systems is that they tend to run the same software over and over, as defined by their application. Based on the application-specific nature of such systems, a recent line of work has proposed application-specific power and energy reduction techniques that identify hardware resources (e.g., gates) in a processor that cannot be exercised by the application running on the processor and eliminate the power used to support those resources [12–14]. However, such application-specific optimizations can only be safely applied if an analysis technique can guarantee that the application running on the processor will never use the resources for any possible execution of the application, for any inputs. Eliminating gates or power for resources that could be used by the application could lead to incorrect execution of the application. For example, power gating a gate that was incorrectly identified as "unused" but is actually exercised by an application can result in the application producing incorrect outputs or crashing [13]. Given the need for guarantees and the inability to achieve such guarantees through input-based application profiling, recently-proposed application-specific power management techniques rely on a symbolic simulation [14, 15] of the application on the

processor hardware to identify hardware resources that are guaranteed to not be used across all possible executions of an application. By propagating symbols that represent unknown logic values for all inputs to an application, it is possible to determine all possible hardware resources that could be used by the application in an input-independent fashion [14]. Recent work has demonstrated that the input-independent activity profiles generated by such a symbolic simulation of an application running on a processor can be leveraged to identify worst-case timing, power, and energy characteristics for a low-power system and to eliminate power used by resources that the system's captive application is guaranteed to never use [12–14, 16].

This approach is sound and characterizes the application based on the instructions in the application binary. However, some of the program semantics are lost because of optimization techniques used in prior works. To handle the large number of possible execution paths for applications with complex control structures, prior work maintains *conservative* states at PC-changing instructions [16]. A conservative state encompasses a superset of all observed states every time the simulation re-visits the PC. If a state is a sub-state of the conservative state maintained at the PC, that state has already been simulated, and execution from the state can be terminated.

The conservative state based approach allows analysis to complete sooner, but suffers from the pessimism of marking too many gates as exercisable, potentially leaving significant benefits on the table. This is due to the nature of conservative state construction, where states are merged by replacing locations that are different with Xs, representing unknown logic values; thus, the number of states represented by the resulting super-state can be exponentially more than the number of states used to generate the conservative state. This can lead to covering states that are not possible in the original application [17]. In this work, we characterize the behavior of an application by analyzing the binary to determine constraints, e.g., bounds of a particular memory element. Such bounds can be used to *constrain* the value of the memory element from being overly pessimistic (i.e., containing too many Xs), leading to fewer gates marked as exercisable and reduced simulation times [17].

Constraints from application software help us to optimize an existing processor design for better energy efficiency. However, design optimization for energy efficiency

should consider not only design-level optimization but also architecture-level optimization. Some architectures may suit one application better than another. For example, an architecture that contains a hardware multiplier may perform multiplication operation in one cycle but consume more energy than an architecture that uses repeated addition to perform multiplication. Depending on the energy requirements, we may chose to add or remove the hardware multiplier for the architectures. In another example, a pipelined design has shorter critical paths than a non-pipelined version of the same design allowing the gates to have a lower drive strength. In contrast, increasing pipeline stages could cause energy overhead from both inserted registers and clock distribution [18]. Depending on factors that are dominating, either the pipelined or the non-pipelined version of a design could be more energy-efficient. Clearly, if energy efficiency is the goal, the processor architecture must also be optimized. Given the wide variety of architectures for embedded processors, there are a lot of choices to consider. This brings in new challenges. We need a tool that analyzes application behavior on any architecture. Despite the significant potential of application-specific design and optimization techniques, applicability has been limited, since the symbolic co-analysis tools developed in previous works were developed for a single processor (openMSP430) [13,14,16,19], and extending them to analyze and optimize other processor designs or architectures requires the challenging and time-consuming task of developing a new custom simulation tool for each new design. This simulation approach is not scalable, especially for industry, as each application may use a different design, and it is infeasible to write a custom simulation tool for each design. So, we built a design-agnostic simulation tool that can handle any design, technology or architecture. For this, we use iverilog - an open source synthesis and simulation tool. In this work, we discuss how we restructured iverilog to allow us to perform application specific processor optimization on any given processor-application pair [20].

Another challenge that the architecture choices bring is the enormous design-space to evaluate. A processor's architecture can be different depending on microarchitectural features such as register-file, memory, adders, multipliers, and many more. In addition to number of microarchitectural features, the choice of the feature implementation also adds to exploration. For example, a multiplier can have a and-gate, nand-gate, or mux-based implementation. In another example, an adder can be implemented as a

ripple-carry adder, carry-save adder, or several other options. All these options exponentially increase the architectural parameter space. Enumerating and exploring this search space includes applying design automation techniques such as synthesis, placement, and routing of the design in addition to symbolically evaluating the application on the hardware. Though we have a generic tool to evaluate any architecture, the run time to evaluate all the possibilities is prohibitively expensive. Moreover, the impact of application-specific hardware optimizations on the energy profile of an architecture is non-trivial. There is no deterministic way to identify an architecture that is most energy-efficient without performing the hardware optimizations and then evaluating the application on the optimized design. Considering the enormous design space, significant simulation time and non-trivial impact of the hardware optimizations, we rely on Machine Learning (ML) to help us minimize the number of architectural options that we must evaluate. We present a tool that takes into account the impacts of application-specific optimizations on different architectural features and predicts near-optimal architecture that best suits the application in terms of energy efficiency. We evaluate the top few predicted architectures using our generic tool and pick the design with the highest energy efficiency.

With our focus on generating bespoke hardware for an application of interest, and given the prominence of data, privacy, and security in our data-driven world, we sought to explore bespoke processors for Secure Multi-Party Computation (MPC). The field of MPC has gained significant attention in recent years due to its ability to allow multiple parties to collaboratively compute a function on their private data while preserving their privacy. MPC finds a wide range of applications in fields such as finance, healthcare, and social media, where data privacy is of utmost importance. However, one of the primary challenges faced by MPC is the high communication overhead among parties, which can lead to significant performance degradation. Another major challenge for the wide adoption of MPC as a way to develop applications involving private data is the requirement of in-depth domain-specific knowledge, making the technology difficult to access and challenging to deploy. To address these challenges, we sought to apply bespoke optimization techniques to customize a general-purpose processor (GPP) for MPC applications. However, because the computational paradigm for MPC is radically different than the GPP paradigm, with communication heavily interleaved in the computation,

a bespoke customization of a GPP fails to overcome the bottlenecks inherent in MPC applications sufficiently to make MPC a feasible solution for most applications. With this understanding, we sought to leverage our expertise in using application-based constraints to generate efficient application-specific processors to create a bespoke domain-specific processor for MPC applications. Designed with an MPC-optimized instruction set architecture (ISA), our bespoke domain-specific MPC processsor becomes a new *template* from which bespoke MPC designs can be generated for specific MPC applications. Optimizing the processor ISA and microarchitecture for MPC allowed us to achieve $\sim 20000\times$ improvement in performance and energy efficiency compared to a state-of-art GPP-based design optimized for MPC.

The performance of our bespoke domain-specific MPC processor can enable the benefits of MPC to be extended to new application areas; however, this will only happen if we also overcome the need for domain-specific expertise in order to develop software for MPC applications. To this end, we have developed a software toolchain for MPC processors that provides a simplified python programming interface and eliminates the need for the MPC application developer to have in-depth knowledge of MPC. We also provide a compiler, assembler, and ISA that are specifically tailored to MPC applications, giving special importance to communication, ordering of instructions, and overall performance. Both the software toolchain and hardware architecture for our bespoke MPC processor are compatible with bespoke optimization, i.e., the processor can be customized to support only the instructions or instruction sequences in a specific MPC application, and the software toolchain can compile and assemble applications for bespoke MPC processors.

This dissertation is organized as follows. Chapter 2provides the background for co-analysis techniques used to devise application-specific hardware optimizations. Chapter 3 [17] introduces application-based software constraints and discusses the means and impact of using them for application-specific hardware optimizations. Chapter 4 [21] presents a generic tool that performs application-specific analysis on any design. Chapter 5 emphasizes the importance of modifying architecture for energy-efficiency and shows how Machine Learning can help contain the design-space exploration when several architectural parameters are considered. Chapter 6 presents a hardware and software co-design approach to develop bespoke processors for Secure Multi-Party Computation.

Chapter 7 concludes the thesis and discusses a few future research directions.

Chapter 2

Background

This chapter provides background information on co-analysis techniques used to devise application-specific hardware optimizations.

2.1 Symbolic Simulation Based Hardware Software Co-analysis

Application-specific nature of emerging ultra-low-power systems [2–11] provides the opportunity to make application-specific optimizations on the processor used in such systems. A recent line of work [12–14] has proposed application-specific power and energy reduction techniques that identify hardware resources (e.g., gates) in a processor that cannot be exercised by the application running on the processor and eliminate the power used to support those resources. However, such application-specific optimizations can only be safely applied if an analysis technique can guarantee that the application running on the processor will never use the resources for any possible execution of the application, for any inputs. Eliminating gates or power for resources that could be used by the application could lead to incorrect execution of the application. For example, power gating a gate that was incorrectly identified as "unused" but is actually exercised by an application can result in the application producing incorrect outputs or crashing. Given the need for guarantees and the inability to achieve such guarantees through input-based application profiling, recently-proposed application-specific power management techniques rely on a symbolic simulation [15] of the application on the

processor hardware to identify hardware resources that are guaranteed to not be used across all possible executions of an application. By propagating symbols that represent unknown logic values for all inputs to an application, the work in [12–14] characterizes the gate-level activity of a processor executing an application for all possible inputs. During the gate-level simulation, the simulator sets all inputs to Xs, which are treated as both 1s and 0s. In each simulation cycle, gates where an X propagated are considered as toggled, since some input assignment could cause the gates to toggle. The set of gates that have toggled during the simulation determines the possible hardware resources that could be used by the application for any application input.

Symbolic simulation is an effective methodology to analyze a design for all application inputs using a single simulation. However, replacing application inputs with symbols makes it challenging to handle input dependent control flow paths. For example, if an X propagates to the PC, it is unclear how execution must proceed. The work in [12–14] branches the execution tree and simulates execution for all possible branch paths, following a depth-first ordering of the control flow graph. Since this naive simulation approach does not scale well for complex or infinite control structures which result in a large number of branches to explore, the work in [16] employed a conservative approximation method that allows the analysis to scale for arbitrarily-complex control structures while conservatively maintaining correctness in identifying exercisable gates. For the approximation to work, [16] generates and maintains conservative states.

2.2 Conservative State

A conservative state is defined as the gate-level state of a processor that conservatively represents multiple observed states for each control-flow changing instruction of the application. For example, a conservative state with a register value of XX can represent four different states with the same register possessing one of the 00,01,10,11 values as shown in Figure 2.1. The approximation works by tracking the most conservative gate-level state that has been observed for each PC-changing instruction (e.g., conditional branch). When a branch is re-encountered while simulating on a control flow path, simulation down that path can be terminated if the symbolic state being simulated is a substate of the most conservative state previously observed at the branch (i.e., the

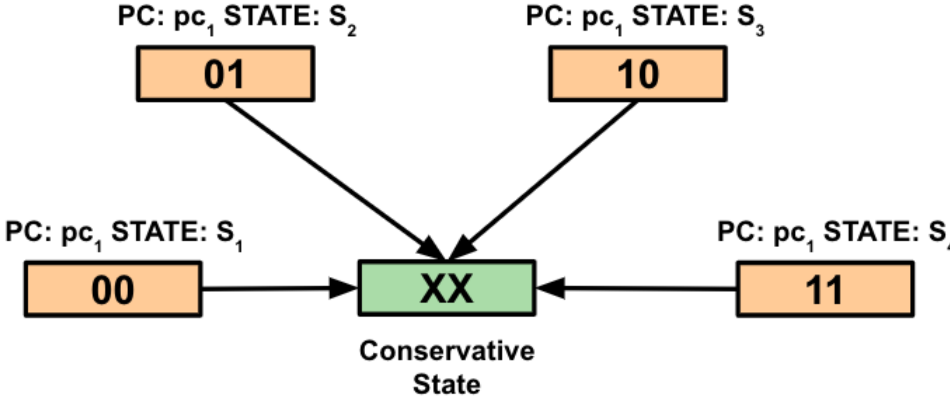


Figure 2.1: Example of conservative state representing register values at different execution states for the same PC.

states match or the more conservative state has Xs in all differing variables), since the state (or a more conservative version) has already been explored. If the simulated state is not a substate of the most conservative observed state, the two states are merged to create a new conservative symbolic state by replacing differing state variables with Xs, and simulation continues from the conservative state. This conservative approximation technique allows gate activity analysis to complete in a small number of passes through the application code, even for applications with an exponentially-large or infinite number of execution paths.

This conservative approximation method is effective. However, it still treats the application as a black box, and hence, suffers from the pessimism of marking too many gates as exercisable, potentially leaving significant benefits on the table. This is due to the nature of conservative state construction, where states are merged by replacing locations that are different with Xs; thus, the number of states represented by the resulting super-state can be exponentially more than the number of states used to generate the conservative state. This can lead to covering states that are not possible in the original application. In the next chapter, we discuss how application information can help reduce the conservativeness of this approximation method.

Chapter 3

Constrained Conservative Symbolic Hardware-Software Co-analysis

Conservative state based symbolic hardware-software co-analysis allows the gate activity analysis to complete in a small number of passes through the application. However, the conservative approximation results in exploring execution paths that are not actually possible for the application. In this chapter, we demonstrate the concept of conservative state using an example and illustrate its limitations. We also show how application information can be exploited to reduce some of the over-approximation.

3.1 Conservative State Limitation

Conservative states are generated from previous simulated states by replacing locations that are different with Xs. The idea is to represent all simulated states with one conservative state. Conservative states allow terminating a simulation when the simulation encounters a previously simulated branch and the simulation state is a substate of the most recent conservative state for the corresponding PC. We illustrate the behavior and limitation of Conservative states using an example.

The example code in Figure 3.2 (compiled from C-code in Figure 3.1) represents a

```
int p=2, q=*val;int i;
for(i=16; i>0; i--){
        ...
        if (p < q){
            p += q;
        }
} // i > 0; i--
return;
```

Figure 3.1: Example C program.

```
1. mov #16, r5
2. mov #2, r13
3. mov &200, r14
loop:
4.    ...
5. cmp r13, r14
6. jnc then
7. add r14, r13
then:
8.dec r5
9.jnz loop
10. ret
```

Figure 3.2: Compiled MSP430 program.

simple subroutine that updates an internal variable (represented by `r13` (`p`)), based on an external value (represented by `r14` (`q`)), over 16 iterations (tracked by `r5` (`i`)). The first section of the code (red) initializes the registers `r5`, `r13`, and `r14`. The next two sections (blue and yellow) are the loop body, where `r13` is compared against `r14`. If `r14` $\geq$ `r13`, line 7 is executed to increase `r13` by `r14`. Otherwise, simulation iterates again, after decreasing the loop counter (`r5`) in the next section (green). After exiting the loop, we `return` from this subroutine.

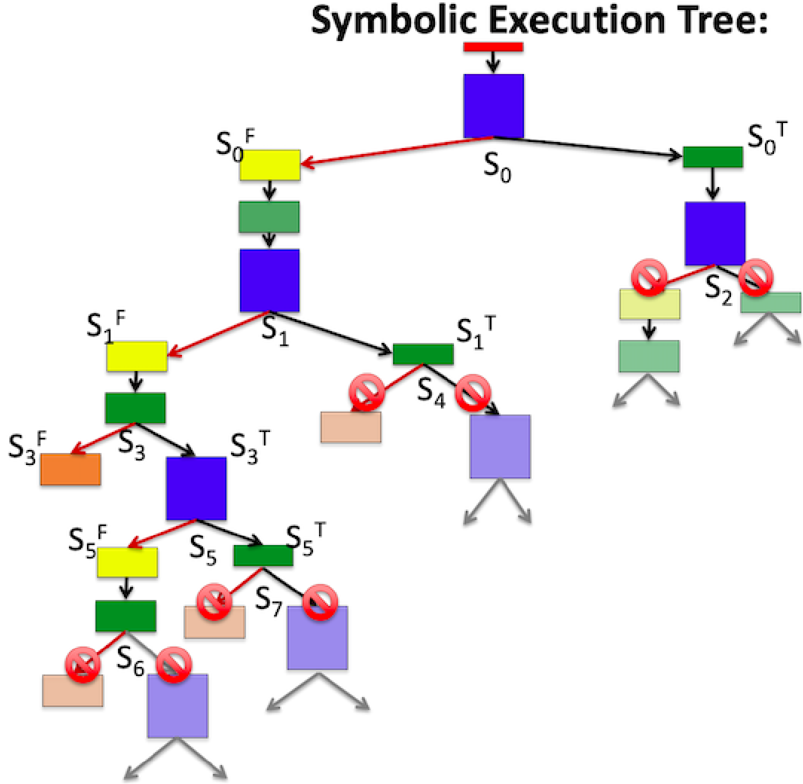
To get the gate activity of the example code, the symbolic simulation replaces the external value (represented by `r14` (`q`)) with Xs. Figure 3.3 shows the execution tree of conservative state based symbolic simulation and the values of two registers `r13` and `r5` at various states that the processor reaches during execution. The simulation starts in the red block and reaches the end of the blue block. Since `r14` contains Xs, the subsequent jump `jnc`'s path is inconclusive, and an X propagates to the PC. We split the simulation to execute both branch paths – the yellow block and the green block. The state of the processor at the end of the blue block is represented as S_0, and the states of the processor at the start of false and true paths are represented as S_0^F and S_0^T,

respectively. The same convention is used for the rest of the states in the tree. Each state in the table contains two rows for the values of the registers `r13` and `r5`. The upper row represents the value of the register observed when the simulation reaches the corresponding point in the execution tree. The lower row represents the conservative value computed by merging this value with the previous conservative state observed at this point.

Simulation proceeds using this conservative value instead of the observed value. One example of conservative approximation is that of register `r5` for state S_1. Since S_1 and S_0 correspond to the same PC, we build a conservative state to represent both the states S_1 and S_0 when we simulate down S_1; this is achieved by replacing the values that differ between the two states with Xs. In the case of `r5`, the two states differ in the least significant 5 bits, which are replaced by Xs to represent both the states. This *X-ification* of the states leads to skipping execution of several states downstream and thus a faster completion of application analysis.

However, the conservative over-approximation of `r5` at S_1 represents not only the two states merged but also all 32 states representable by varying the lower 5 bits of `r5`. Therefore, when we execute the instruction `dec r5` in the green block just before state S_3, the value 16'bXXXXX can represent 32 different values, including 16'b0 and 16'b1. Decrementing 16'b0 by 1 results in 16'b1111111111111111 (two's complement arithmetic), while decrementing 16'b1 by 1 results in 16'b0. To represent both these states, `r5`'s value becomes 16'bXXXXXXXXXXXXXXXX. Unfortunately, this represents all the 2^{16} possible values for a 16-bit number. However, from the example code, we know that `r5` only actually assumes values between 0 and 16 and the code only toggles lower order five bits of the register `r5`. By propagating the value of 16'bXXXXXXXXXXXXXXXX for `r5`, the conservative state based symbolic simulation also toggles the upper ten bits of `r5`. Considering the fanout gates of the upper ten bits of `r5`, the simulation exercises many more gates in the processor than necessary.

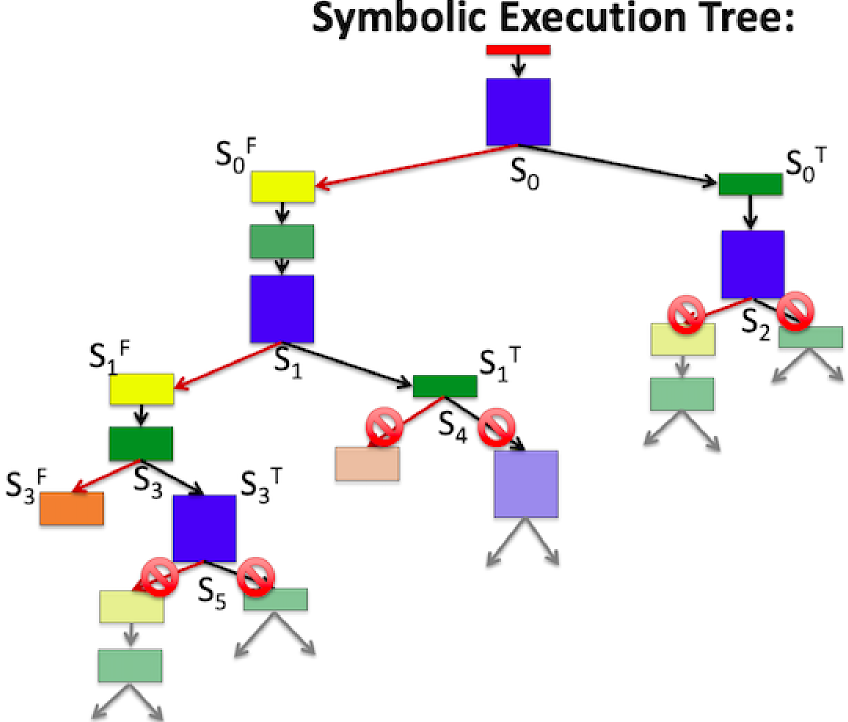
In our work, constrained conservative state symbolic hardware-software co-analysis, we translate constraints on variables at the software level to constraints on memory elements in the processor-memory system. In this example, since $0 \leq$ `r5` < 17, we constrain the value of `r5` to 16'b00000000000XXXXX, preventing the unnecessary propagation of Xs. Figure 3.4 shows the execution of the example code in Figure 3.2 using



Conservative States:

State	PC	r13	r5
		R13(c_state)	R5(c_state)
S_0	6	0b0000000000000010	0b0000000000010000
		0b0000000000000010	0b0000000000010000
S_1	6	0bXXXXXXXXXXXXXXXX	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_3	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_4	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_5	6	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_6	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
S_7	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX

Figure 3.3: Conservative state-based scalable symbolic co-analysis can analyze applications with infinite loops and input-dependent branches by simulating conservative states that capture the activity of multiple possible states.



Constrained Conservative States(CCS):

State	PC	r13	r5
		R13(cc_state)	R5(cc_state)
S_0		0b0000000000000010	0b0000000000010000
	6	0b0000000000000010	0b0000000000010000
S_1	6	0bXXXXXXXXXXXXXXXX	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_3	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_4	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	**0b00000000000XXXXX**
S_5	6	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0b00000000000XXXXX

1 <= r5 < 17

Figure 3.4: Constraining memory elements based on bounds from the software level reduces pessimism in estimating the number of gates marked as exercisable and also reduces the number of paths that need to be explored.

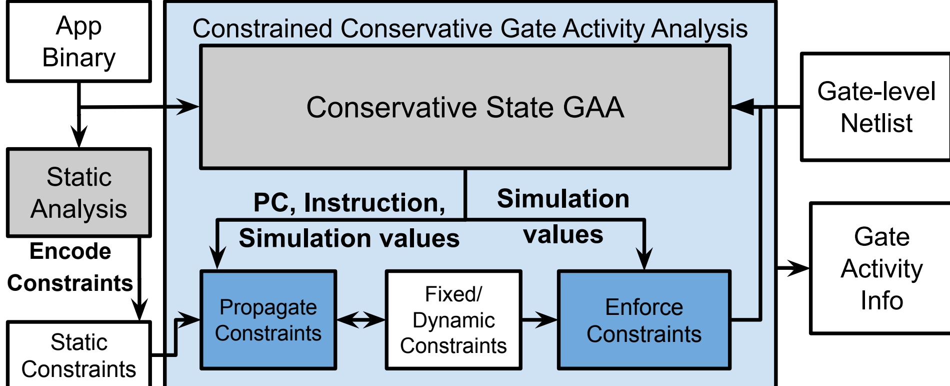


Figure 3.5: Methodology for CCS

constrained conservative state symbolic execution. This not only reduces the number of gates toggled; it also reduces the number of execution paths simulated, leading to faster convergence.

3.2 Proposed Work

In this section, we explain Constrained Conservative State Symbolic Hardware-Software Co-Analysis. Our co-analysis tool (see Figure 3.5) is based on the observation that certain constraints on variables at the software level are lost when the application is simulated at the gate-level, leading to overly pessimistic estimates of the hardware resources (i.e., gates) needed to execute the application. We translate software-level constraints to the gate level in three steps. First, we encode high-level program constraints as constraints on the operand values of static instructions. Our tool generates these constraints from a pattern-based static analysis of the application binary. Second, these encoded constraints are loaded into the conservative symbolic simulator and propagated from source operands to destination operands during simulation. Third, when operands containing Xs are updated by an instruction, encoded and propagated constraints are applied so that the operands' symbolic values observe the constraints. Pseudocode of our implementation is shown in Algorithm 1 and Algorithm 2. Changes to the conservative symbolic co-analysis in Algorithm 1 are presented in **red**. In the following subsections, we explain each step in greater detail.

Algorithm 1 Constrained Conservative State Symbolic Co-analysis

```
1. Procedure GateActivityAnalysis(app_binary, design_netlist)
2. Initialize all memory cells and all gates in design_netlist to X
3. Load app_binary into program memory
4. Propagate reset toggle signal
5. s ← State at start of app_binary
6. Symbolic Execution Tree T.set_root(s)
7. Unprocessed execution points queue, U.push(s)
8. C.init() // Initialize conservative system state map
9. C_T.load_constraints() // Load Static constraints map
10. while U != ∅ do
11.    e ← U.pop()
12.    if e.isConditinalBranch() and e.PC ∈ C then
13.       a ← C.getState(e.PC)
14.       if e.isConservativeSubstateOf(a) then
15.          continue
16.       else
17.          e ← buildConservativeState(a, e)
18.          C ← C.update(e.PC, e)
19.       end if
20.    else if e.isConditionalBranch() then
21.       C ← C.add(e.PC, e)
22.    end if
23.    while e.nextPC != X and !e.END do
24.       e.setInputsX() // set all peripheral port inputs to Xs
25.       e′ ← propagateGateValues(e) // perform simulation for this cycle
26.       if e′.aboutToCommit() then
27.          // instruction will be committed in the next cycle
28.          c_t ← getConstraints(C_T, e′.PC)
29.          e′ ← propagateConstraints(e′, c_t) // transfer constraints, source to destination
30.          e′ ← enforceConstraints(e′, c_t)
31.       end if
32.       e.annotateGateActivity(e, e′) // annotate tree point with activity
33.       e.addNextState(e′) // add to execution tree
34.       e ← e′ // process next cycle
35.    end while
36.    if e.nextPC == X then
37.       for all a ∈ possibleNextPCVals(e) do
38.          e′ ← e.updateNextPC(a)
39.          U.push(e′)
40.          T.insert(e′)
41.       end for
42.    end if
43. end while
```

3.2.1 Encoding Constraints From Binary

In order to constrain simulation values, our tool must know what memory element's values should be constrained, what its valid set of values are, and at what execution points those constraints are valid. As shown in Figure 3.6, our tool takes value bound constraints (e.g., 0 to 17) on instruction operands (e.g., r5) for specific instructions (e.g., the `mov`, `dec`, and `jnz` instructions at PCs 3, 9, and 10, respectively).

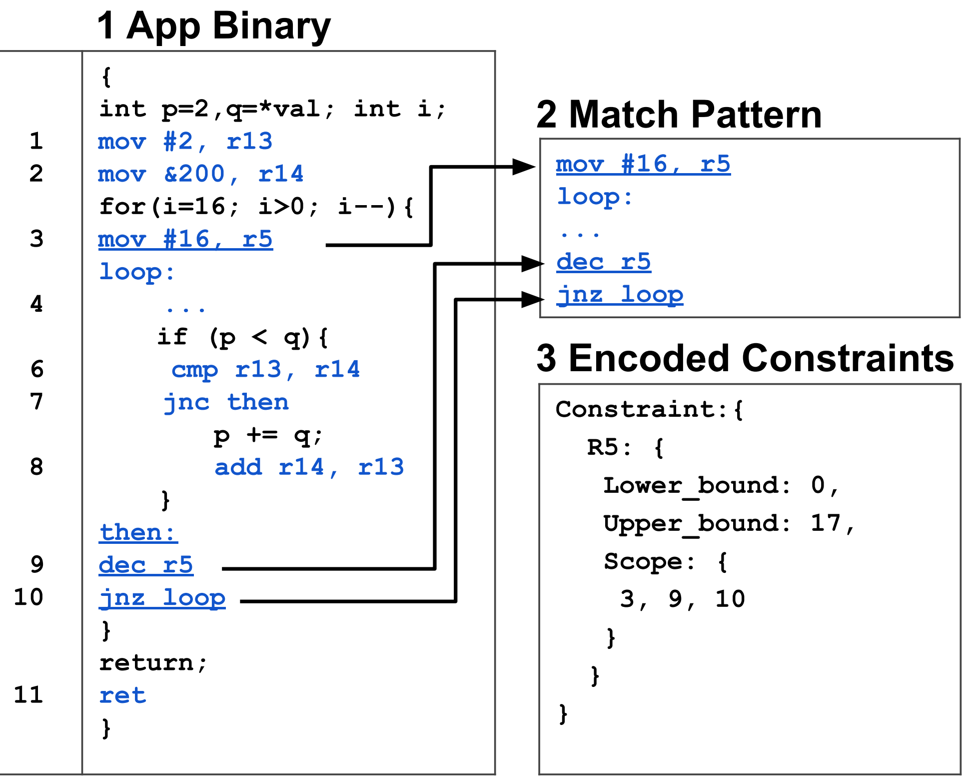


Figure 3.6: Example of constraint encoding during static analysis of the application binary.

An example instruction pattern is shown in Figure 3.6. Many possible static analyses at different abstraction levels, from C compiler to binary analysis, could be used to generate constraints, with varying trade-offs of coverage and precision [22–24]. For our work, we chose to use a pattern-based binary analysis approach where we map known binary patterns resulting from high-level program structures (e.g., loops and if statements) into constraints (e.g., register holding a loop iterator is bounded between its initialization and termination values at loop boundaries). We have identified nine such patterns involving different types of loops and nested loops. Note that for pattern-based analysis, the relevant patterns can depend on compiler options. Our library of patterns covers the most common patterns observed in our benchmark set (see Section 4.4).

3.2.2 Propagating Constraints

Once we encode all the constraints, we load them into the co-analysis tool as `Fixed` (i.e., immutable) constraints on operands (i.e., register and memory values) at specific static instructions, and we start symbolic co-analysis. During co-analysis, we intercept every instruction when it is about to be committed in the processor pipeline, read the constraints on the instruction's source operands, and update the constraint on the destination operand if that operand does not have a `Fixed` constraint at the current PC. This updating creates a `Dynamic` constraint for the memory element.

Consider the instruction `mov #2, r13`, with `r13` having no constraint before the instruction is executed. At the end of the execution of the instruction, we will have a constraint on `r13` as $2 \leq \texttt{r13} < 3$, representing its constant value. Consider another instruction, `add r5, r13`, with constraints on `r5` as $1 \leq \texttt{r5} < 17$ and on `r13` as $2 \leq \texttt{r13} < 3$. Since the value of `r5` does not contain Xs (it is 16), the constraint of `r13` is updated by adding `r5`'s value (16) to the lower and upper bounds of `r13`'s constraints to produce the constraint: $18 \leq \texttt{r13} < 19$. However, if the value of `r5` is 16'bXXXXX, the constraint on `r13` is updated to $3 \leq \texttt{r13} < 20$, by adding the lower bounds and the upper bounds of the two constraints, respectively. This ensures that constraints are as tight as possible while encompassing all possible values.

3.2.3 Enforcing Constraints

Encoding and propagating constraints ensures that values of registers or memory locations that are constrained cannot go out of bounds of these constraints. To ensure this, we monitor all register and memory location values for changes during simulation. Whenever a register or a memory location is modified, we check its value against any constraint it has. If the value of the register or memory location could be out of bounds of the constraint, we enforce the constraint on the register or memory location by modifying its value appropriately. Our technique ensures that enforcing constraints does not eliminate exploration of any reachable states for a given application. A formal proof is presented in Section 3.3.

In addition to constraining memory and register values, it is important to ensure that memory *addresses* do not go out of bounds. In an indirect addressing mode, if the register holding the memory address contains Xs, there are several possible addresses that could be accessed. In such a case, the constraint on the register restricts the number of possible memory locations. While performing memory reads, all possible memory addresses (defined by the constrained conservative value) are read, and a conservative value is generated out of data read from memory. This value is sent to the data bus and used by the instruction. Similarly, while handling a memory write, both the address and the value could have Xs. In this case, we first resolve the constraint on the address by identifying the permissible locations for the element, based on the constraint and the value of the address. We then generate conservative values and update the constraints at all the resolved addresses. For instance, consider the instruction `mov r5, -5(r6)`. Assume that both `r5` and `r6` contain Xs. To handle proper execution of this instruction, we first obtain the constraint for `r6` and adjust the address constraint for `-5(r6)` according to the offset (i.e., `Lower_bound -5(r6)` $\leftarrow$ `Lower_bound (r6) - 5`) and `Upper_bound -5(r6)` $\leftarrow$ `Upper_bound (r6) - 5`). Then, for each address represented by `-5(r6)`'s value in the simulator (the value with the Xs), we check if the address is in the range of the constraint (i.e., `Lower_bound -5(r6)` $<$ address $<$ `Upper_bound -5(r6)`). For the addresses that are in the bound of the constraint, we write the conservative value of `r5` combined with the existing memory value to the locations pointed by the resolved addresses. This algorithm is presented in Algorithm 2.

Algorithm 2 Constraint Enforcement

```
1.  // e : Execution state of the processor
2.  // c_t : Constraint
3.  Procedure enforceConstraints(e, c_t)
4.  if e.isOutputOutOfBounds(c_t) then
5.      if e.isMemoryOp() then
6.          e ← handleMemoryEnforcement(e, c_t)
7.      else
8.          e.dstRegVal ← genConstrainedVal(e.dstRegVal, c_t)
9.      end if
10. end if
11. return e

12. //e : Execution state of the processor
13. // c_t : Constraint
14. Procedure handleMemoryEnforcement(e, c_t)
15. if containsX(e.memAddress) then
16.     for all addr ∈ possibleAddresses(e.memAddress) do
17.         if isAddressInBounds(addr, c_t.addressConstraint) then
18.             if e.memOperation == read then
19.                 val ← generateConstrainedConservativeVal(val,
                    e.dMemory[addr], c_t.valConstraint)
20.             else if e.memOperation == write then
21.                 e.dMemory[addr] ← generateConservativeVal(e.val, e.dMemory[addr])
22.             end if
23.         end if
24.     end for
25.     if e.memOperation == read then
26.         e.dataBus.put(val)
27.     end if
28. else
29.     addr ← e.memAddress
30.     if e.memOperation == read then
31.         val ← e.dMemory[addr]
32.         e.dataBus.put(val)
33.     else if e.memOperation == write then
34.         e.dMemory[addr] ← e.val
35.     end if
36. end if
37. return e
```

3.3 Proof of CCS Correctness

Theorem 1 (Application Execution State Coverage)**.** *Given a constraint* $\boldsymbol{c}$ *and an element (register/memory address)* $\boldsymbol{e}$*, enforcing* $\boldsymbol{c}$ *on* $\boldsymbol{e}$ *at a PC* $\boldsymbol{p}$ *does not eliminate exploration of any reachable states for application* $\boldsymbol{A}$*.*

Proof. Let $\boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_n}$ be consecutive conservative states generated at PC $\boldsymbol{p}$ by the Conservative State (**CS**) approach. By definition of conservative state, $\boldsymbol{S_1} \subset \boldsymbol{S_2} \subset \ldots \subset \boldsymbol{S_n}$. Let $\boldsymbol{S_i}$ be the first state where $\boldsymbol{e}$ violates $\boldsymbol{c}$. Thus, $\boldsymbol{S_i}$ covers all executions leading to $\boldsymbol{p}$ that have been explored until the ith encounter of $\boldsymbol{p}$. I.e., for all states before

Table 3.1: Benchmarks

Embedded Sensor Benchmarks [28]
mult, binSearch, div, inSort, tea8, rle, tHold, intAVG, intFilt
EEMBC Embedded Benchmarks [29]
AutoCorr, convEn, FFT, Viterbi
Complex Benchmarks
MergeSort , graph500 [30], highCC

$\boldsymbol{S_i}$ $(\boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_{i-1}})$, the Constrained Conservative State (**CCS**) approach and **CS** are identical. Since $\boldsymbol{S_i}$ violates $\boldsymbol{c}$, it necessarily covers some states that are not reachable by $\boldsymbol{A}$. Constraining $\boldsymbol{e}$ using $\boldsymbol{c}$ generates $\boldsymbol{S'_i}$ such that $\boldsymbol{S'_i}$ covers all possible values that $\boldsymbol{e}$ can assume in $\boldsymbol{A}$; only unreachable states are eliminated through the application of $\boldsymbol{c}$. Thus, continuing the simulation from $\boldsymbol{S'_i}$ will explore all valid states that are reachable by $\boldsymbol{A}$. □

3.4 Evaluation

We perform evaluations on a silicon-proven openMSP430 [25] processor, synthesized, placed and routed in TSMC 65GP (65nm) technology using Synopsys Design Compiler [26] and Cadence EDI System [27]. The processor was implemented for an operating point of 1V and 100MHz. We implemented our constrained conservative state-based scalable symbolic co-analysis in a custom gate-level simulator that was built in-house in C++. We also developed a custom static binary analysis tool in Python for encoding constraints. The static constraints were stored in a JSON file and fed to the custom gate-level simulator, which the simulator uses for Propagation and Enforcement. We show results for all benchmarks from [28], all EEMBC benchmarks [29] that fit in the program memory of our processor, as well as complex and recursive benchmarks[1] designed to stress-test the scalability of our symbolic hardware-software co-analysis technique with complex control structures not found in the rest of our benchmarks (Table 3.1). Experiments are performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz operating frequency, 64GB RAM).

To illustrate the benefits of our proposed technique for symbolic co-analysis, we

[1]MergeSort is a recursive sorting algorithm. graph500 runs BFS on a graph. highCC (high Cyclomatic Complexity) is a synthetic benchmark that uses cyclic array accesses to alter the control flow of the application and has 16^{32} possible control flow paths.

Table 3.2: Constrained conservative state symbolic co-analysis reduces analysis time compared to naive and conservative state-based co-analysis and enables analysis of applications with complex control structures.

Benchmark	**Analysis Time (Number of Simulation Cycles)**				
	Naive	**Consv.**	**CCS**	**%Reduction (w.r.t. Naive)**	**%Reduction (w.r.t. Consv.)**
div	∞	186	178	-	4.30
intAVG	∞	337	329	-	2.37
rle	∞	7431	5951	-	19.92
rle_small	25496	6495	2153	91.56	66.85
binSearch	100468	9994	1551	98.46	84.48
tHold	20520	2615	1986	90.32	24.05
inSort	∞	22205	12120	-	45.42
inSort_small	24427	9106	5089	79.17	44.11
Viterbi	∞	69265	26389	-	61.90
MergeSort	∞	104574	16093	-	84.61
graph500	∞	185341	79663	-	57.02
highCC	∞	116290	80276	-	30.90

Table 3.3: Use of constraints reduces the number of explored symbolic execution paths.

Benchmark	**Symbolic Execution Paths**				
	Naive	**Consv.**	**CCS**	**%Reduction (w.r.t. Naive)**	**%Reduction (w.r.t. Consv.**
div	∞	9	7	-	22.22
intAVG	∞	15	13	-	13.33
rle	∞	129	101	-	21.71
rle_small	504	113	33	93.45	70.80
binSearch	2048	91	41	98.00	54.95
tHold	460	247	39	91.52	84.21
inSort	∞	121	67	-	44.63
inSort_small	476	115	65	86.34	43.48
Viterbi	∞	771	291	-	62.26
MergeSort	∞	1453	235	-	83.83
graph500	∞	1350	1124	-	16.74
highCC	∞	1604	756	-	52.80

Table 3.4: Use of constraints reduces the number of gates identified as exercisable.

Benchmark	**Exercisable Gates Identified**				
	Naive	**Consv.**	**CCS**	**%Increase (w.r.t. Naive)**	**%Reduction (w.r.t. Consv.)**
div	N/A †	3627	3566	-	1.68
intAVG	N/A †	3675	3648	-	0.73
rle	N/A †	4488	3759	-	16.24
rle_small	3185	4487	3740	17.43	16.65
binSearch	3065	3454	3424	11.71	0.87
tHold	2893	3530	3368	16.42	4.59
inSort	N/A †	5406	3518	-	34.92
inSort_small	3134	5418	3523	12.41	34.98
Viterbi	N/A †	5449	5449	-	0.00
MergeSort	N/A †	5134	4294	-	16.36
graph500	N/A †	5988	5987	-	0.02
highCC	N/A †	4007	3558	-	11.20

† Since these simulations did not finish, naive simulation would be forced to report that all 7218 gates of the design might be exercisable.

compare our *constrained conservative state* (CCS) symbolic co-analysis technique (Algorithm 1 **black**+**blue**+**red** text) against the *naive* symbolic co-analysis technique (Algorithm 1 **black** text only) and the state-of-the-art *conservative* symbolic co-analysis technique [16] (Algorithm 1 **black**+**blue** text). We compare analysis time and exercisable gate counts for the benchmarks described in Table 3.1. We show that the constrained conservative approach addresses the limitations of the naive and conservative approaches by yielding an exercisable gate count closer to the accurate naive approach, while also significantly reducing simulation time compared to the state-of-art with minimal overhead.

For benchmarks with simple control flow (i.e., no input-dependent branches), symbolic simulation only needs to consider a single execution path through the program; conservative states are never created, and the conservative and constrained conservative approaches will perform the same simulation as the naive approach. Since the results for these benchmarks (mult, intFilt, tea8, FFT, AutoCorr, convEn) do not show any variation between the simulation approaches and thus cannot be used to compare the techniques, we omit these benchmarks from our results tables due to space limitations. However, we did use these benchmarks to verify that the results for all three simulation approaches are consistent. Furthermore, our constrained conservative approach does not increase the execution time or number of execution paths considered.

3.4.1 Analysis Time

Table 3.2 compares analysis times for performing the symbolic simulation of each benchmark application. We use simulated clock cycles of the openMSP430 processor as a proxy of analysis time that is independent of the host computer's computational capability and load.[2] Constrained conservative analysis achieves the lowest analysis time for all benchmarks by effectively pruning the execution tree to eliminate consideration of already-visited states and states that are precluded by application constraints. For six of the benchmarks, naive symbolic simulation was not able to complete within 24 hours and was eventually killed after using all of our server's memory (64 GB RAM and 125 GB swap). These benchmarks are marked with ∞ in the naive column of Table 3.2.

[2]The overhead introduced by the constrained conservative analysis indicated by red text in Algorithm 1 is between 1.1% and 1.9% per cycle.

Meanwhile, the conservative state approach is able to analyze all of the benchmarks in under an hour. By applying application constraints on top of the conservative approach, CCS reduces analysis time for each benchmark, with a maximum reduction of 84.61% compared to the state-of-art conservative state approach. Applying software constraints to the symbolic simulation keeps conservative values within their legal ranges, significantly pruning the state space and resulting in a more efficient exploration of the application's possible states.

Table 3.3 shows the number of symbolic execution paths each symbolic simulation approach explores (as described in Section 3.1). In the conservative approach, new symbolic execution path subtrees are created at conditional branches and simulated if they have not been previously explored. By constraining the values of registers/memory elements in the processor, the constrained conservative approach reduces the number of symbolic execution paths that must be simulated to completely characterize all possible executions of an application. This significantly reduces analysis time for several applications. For MergeSort, an application with complex input-dependent control flow, the conservative state approach continues simulating symbolic execution paths until all bits of the loop iterator (for the loop that merges two sorted arrays) become Xs for a given recursive step. In the proposed constrained approach, simulation only proceeds until 6 Xs propagate into the loop iterator, since the maximum bound on the loop iterator is 34 (array size). The result is an 84% reduction of the number of symbolic execution paths that are explored and a corresponding 85% reduction in the number of analysis cycles. As processor complexity increases, the state space of the hardware-software symbolic co-analysis increases, and the potential benefits of constraining the symbolic simulation increase. E.g., a 64-bit processor has exponentially more possible states than a 16-bit processor, so the same loop bounds constraint applied to both would eliminate exponentially more states from consideration in a 64-bit processor vs. a 16-bit processor.

3.4.2 Exercisable Gates

Table 3.4 presents the count of exercisable gates reported by the three symbolic simulation approaches. All three approaches guarantee identification of all possible gates that can be exercised by any possible execution of an application; however, the approaches

vary in their overestimation of the exercisable gates due to conservative state approximations. The naive approach does not use conservative states to cover multiple real states, and therefore, provides the most accurate report of the exercisable gate set. However, because naive simulation attempts to simulate all possible states of an application without approximation, naive simulation is not scalable and does not always complete. For some benchmark applications (e.g., inSort and rle), significantly reducing the input size (e.g., to 5 elements) reduces the size of the symbolic execution tree sufficiently to allow the naive approach to finish. We include *small* versions of those benchmarks in the results tables to enable further analysis and comparison of the simulation approaches.[3]

The conservative state approach identifies more exercisable gates than the naive approach. For applications with complex control flow, the overapproximation of the conservative state approach can be significant. The small versions of rle and inSort demonstrate that the conservative approach can significantly increase the number of gates marked as exercisable compared to naive symbolic simulation (e.g., 73% increase in exercisable gates reported for inSort_small). With the proposed constrained simulation, however, there is only a 12% increase in reported exercisable gates for the same application. Applying application constraints to the symbolic states avoids simulation of states that are not actually possible for the application and can significantly reduce the pessimism of applying conservative states to achieve a scalable symbolic simulation.

Compared to the conservative state approach, CCS reports fewer exercisable gates for all benchmarks, except Viterbi where the result is identical, with a maximum reduction of 35% (inSort). The static analysis used in this work generated a maximum of 7 constraints (for graph500) and a minimum of 1 constraint (for div). More sophisticated static analysis techniques may generate more constraints. Nevertheless, our work shows that even applying a small number of constraints can result in significant reduction of exercisable gates and analysis time compared to state-of-art conservative state symbolic co-analysis. The largest benefits come from benchmarks such as inSort, MergeSort, and

[3]Conservative symbolic simulations report slightly more exercisable gates for inSort_small than for inSort. At first, this seems counterintuitive; however, our analysis revealed that a few instructions were different between the two binaries. These instructions cause different gates to be exercised by each of the binaries. We confirmed that the additional exercisable gates in inSort_small trace back to instruction source/destination operand registers. These gates contribute to fewer than 0.2% of the total gates in the processor design and do not change the behavior of the core algorithm in the benchmarks.

rle, that access data using addresses containing Xs. This can potentially cause the address handler in openMSP430 to exercise all the peripherals, since they are in a unified address space. Constraining the addresses avoids this overapproximation of exercisable resources. Although binSearch also accesses data using addresses containing Xs, its structure already limits the number of Xs in addresses during conservative symbolic simulation, since the binSearch algorithm uses a right shift that guarantees that the upper 8 address bits are always zero. This reduces the exercisable gates reported by the conservative state approach for binSearch. Viterbi implements an iterative pointer chasing algorithm that involves many memory-accessing instructions. With the random memory access pattern of the application, the inputs of these instructions are all Xs, causing all the gates in the memory and peripheral path to be presumed exercisable. Constraints do help to restrict the *number* of memory accesses with unknown pointer values, since the accesses are made in a loop, and the loop bound can be determined by static analysis. This significantly reduces analysis time (by 62%) but does not help to reduce the exercisable gate count. Graph traversal in graph500 also involves a pointer chasing random memory access pattern. Similar to Viterbi, reduction in exercisable gates is negligible, but determining loop bounds via static analysis significantly reduces analysis time, by 57%.

Table 3.5: Symbolic simulation approach comparison.

Approach	Precision	Guarantees	Runtime
Naive	🙂	🙂	🙁
Conservative	🙁	🙂	😐
CCS	🙂	🙂	🙂

3.4.3 Final Remarks

In summary, we qualitatively compare the three symbolic simulation approaches in Table 3.5. All three approaches guarantee identification of all possible exercisable gates for *any* possible execution of an application. The naive approach identifies the toggled gates most precisely; however, this approach suffers in simulation runtime, as it attempts to explore all possible execution paths of an application without any approximation. The conservative state-based approach makes symbolic co-analysis practical in

terms of simulation runtime but sacrifices significant precision with overly-conservative representation of simulation states. The constrained conservative approach further improves analysis runtime compared to the conservative approach and also significantly improves precision by applying application-based constraints to the simulation that reduce both the number of symbolic execution paths simulated and the propagation of unknown logic values (Xs) through the netlist.

3.5 Related Work

3.5.1 Static Analysis

Static analysis of programs is a helpful technique used in many works. The work in [23] presents an algorithm that detects infinite loops in program using symbolic execution based static analysis. The work in [22] presents a fast static loop analysis that estimates loop iteration counts and execution frequencies of code elements. Such static loop analyses are useful in compiler optimizations such as loop unrolling, loop tiling, feedback-directed optimizations and many more. Another area where static loop analyses is used is the worst-case execution time(WCET) analysis. [22] elaborates in detail on some of the applications of static loop analysis. The work in [24] uses static analysis to determine iteration domains of syntactic statements in programs. The iterations domains capture the dynamic instances of the statement during the program execution and are used by the program transformations in the polyhedral model and polyhedral code generation. In our work, we perform static analysis on program binary to map binary patterns resulting from high-level program structures (e.g., loops and if statements) into constraints.

3.5.2 Hardware-Software Co-analysis

Co-analysis techniques presented in prior work [12–14] identify all exercisable gates for an application in a processor through symbolic simulation of the application on the processor netlist. Unfortunately, this co-analysis technique cannot analyze applications with complex control flow or infinite loops. To resolve this issue, prior work [12] proposes maintaining conservative states for each PC-changing instruction (e.g., conditional

branch). A conservative state is a state that covers all simulated states observed at a particular PC-changing instruction. An execution path is simulated only if the current state is not a subset of a previously observed conservative state, in which case a more conservative state is created by merging the current state with the conservative state maintained for the PC-changing instruction and continuing simulation from the new conservative state. The conservative approximation technique enables a scalable gate activity analysis that completes in a small number of passes through the application. However, this conservative over-approximation still treats the application as a black box, and hence, suffers from the pessimism of marking too many gates as exercisable, potentially leaving significant benefits on the table. In our work, we proposed a constrained conservative state symbolic hardware-software co-analysis technique that characterizes the behavior of an application by analyzing the binary and determines constraints that *constrain* the value of the memory elements from being overly pessimistic (i.e., containing too many Xs), leading to fewer gates marked as exercisable and reduced simulation times.

3.6 Summary

In this chapter, we proposed a *constrained* conservative state symbolic hardware-software co-analysis technique that applies constraints to symbolic states to reduce the pessimism in marking gates as exercisable. In addition to guaranteeing identification of all possible exercisable gates for an application execution, the proposed technique significantly reduces simulation time and number of symbolic execution paths explored. Compared to the state-of-art analysis based on conservative states, our constrained approach reduces the number of gates identified as exercisable by up to 34.98%, 11.52% on average, and analysis runtime by up to 84.61%, 43.83% on average.

Chapter 4

Design-Agnostic Symbolic Co-analysis Tool

Symbolic co-analysis has proven to be an effective technique for application-specific design optimizations. We further improved the state-of-art symbolic co-analysis technique with software constraints, allowing application information to impact hardware optimizations. Despite the significant potential of application-specific design and optimization techniques, applicability has been limited, since the symbolic co-analysis tools developed in previous works were developed for a single processor (openMSP430), and extending them to analyze and optimize other processor designs or architectures requires the challenging and time-consuming task of developing a new custom simulation tool for each new design. This simulation approach is not scalable, especially for industry, as each application may use a different design, and it is infeasible to write a custom simulation tool for each design. In this chapter, we introduce a general, automated tool for hardware-software co-analysis that can analyze any processor design and enable the benefits of application-specific design and optimization.

4.1 Gate-Level Simulator

Application-specific optimizations are effective because they remove gates that are not exercised for any execution of the application. One important feature of hardware-software co-analysis tools is the ability to run gate-level simulations. Modern gate-level

simulators such as VCS [31] can perform cycle accurate simulations; however, they do not support all features necessary to run hardware-software co-analysis. For example, modern simulators do not support custom propagation of symbols, management of conservative states, and splitting the simulation on observing a particular symbolic signal. In our work, we developed a design-agnostic simulation tool that performs symbolic hardware-software co-analysis with cycle-accurate precision at the gate level. We extend an open-source design synthesis and simulation tool – iverilog – to support symbolic simulations and enable the use of conservative gate-level execution states. In this chapter, we describe how we extended iverilog to support symbolic hardware-software co-analysis for an arbitrary digital design.

4.2 Extending Iverilog For Symbolic Hardware-Software Co-Analysis

Performing the symbolic hardware-software co-analysis of an application on a microprocessor design involves performing a gate-level simulation in which all application inputs are replaced by symbols (X) indicating unknown logic, thus simulating the behavior of the microprocessor for all possible application inputs. When an X is propagated to an instruction that affects control flow (e.g., branch, jump), multiple simulations are spawned to cover all possible execution paths of the application from the instruction. To handle execution path explosion in complex applications, we follow the approach of using a conservative state to represent all execution states observed at the same program counter (PC) [32]. This guarantees coverage of all possible execution states while allowing the simulations to converge. To accommodate symbolic co-analysis, we make the following modifications to iverilog source code.

1) *Monitor critical microprocessor signals*: To identify when X propagates to an instruction that affects control flow, we implement a system task function called `monitor_x()` in iverilog that monitors a list of signals. For example, the signals could be a combination of the ALU flags, like N, Z, C, and V (negative, zero, carry, and overflow) that determine the result of a conditional branch instruction that indicates if a branch is taken.

2) *Save the simulation state*: To cover all possible executions from a branch with

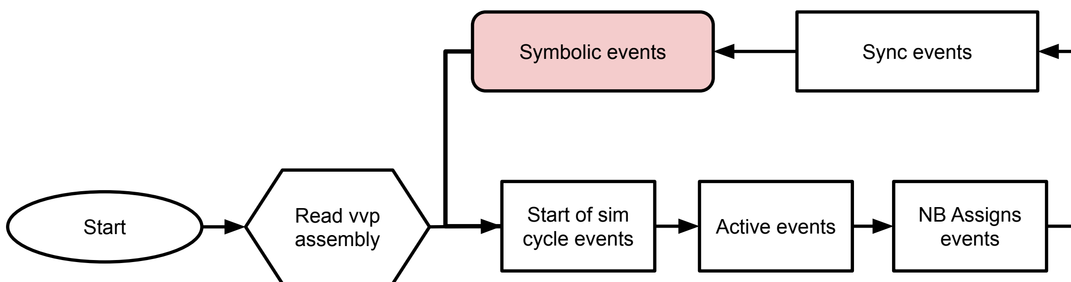


Figure 4.1: We add a new type of event to capture 'symbolic events' in iverilog's event queue. This enables us to monitor control signals for X and halt the simulation when necessary. The VVP engine is a part of iverilog source that executes an iverilog compiled assembly code that is generated from the verilog testbench.

unknown outcome, we first dump the simulation state before the execution of the instruction that affects control flow. The simulation state indicates the state of the microprocessor along with the state of the simulator (e.g., the event queue).

3) *Continue simulation from a saved state*: To simulate all possible executions from the instruction affecting control flow, we make multiple copies of the saved simulation state and modify each copy with the status that allows the microprocessor to take one of the possible executions. We enhance the simulator to read the modified simulation state and continue the simulation from the halted state. For this, we implement another system task called `initialize_state()`.

4.2.1 Iverilog Software Flow Enhancement

iverilog is an event-driven simulator, where a set of events represents a time step. Upon the execution of these events, the simulation time progresses. Events are categorized into five event regions, and each region represents a similar set of events. The event regions are executed in the order shown in Figure 4.1. Since we implement symbolic simulation as a plug-in feature to iverilog, we ensure that our modifications do not affect the existing flow. Therefore, we create a new event region called *Symbolic events* and execute them after the other event regions. Symbolic events includes monitoring control flow signals, halting the simulation when X is detected, serializing and saving the processor and simulator state, and restarting the simulation from a saved state. By

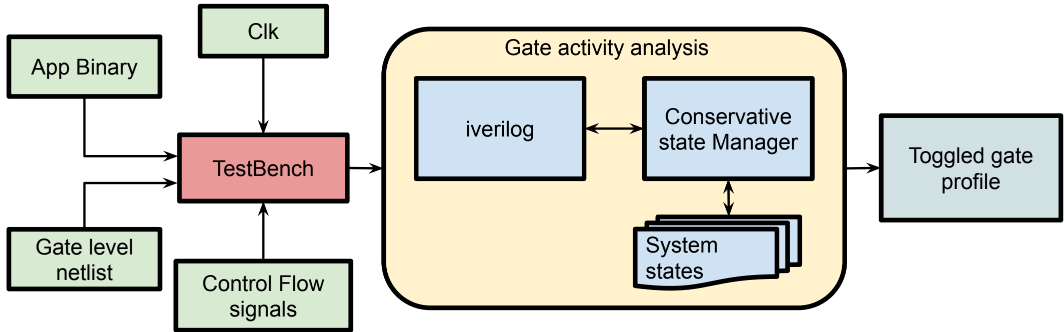


Figure 4.2: Our design-agnostic symbolic co-analysis tool is built on top of iverilog to allow hardware-software co-analysis of any digital design.

executing symbolic events last, we ensure that all events for the time step have already executed. When the simulation restarts, there may be a few events not belonging to the symbolic events region that are executed before initialization. However, the state initialization in the symbolic events region overrides the entire simulator and processor state. This nullifies the effects of any event executed before initialization. As this override occurs only in the first time step, the overhead of this process is minimal.

4.3 Symbolic Hardware-software Co-analysis Using Iverilog

Figure 4.2 illustrates the entire simulation flow of our design-agnostic symbolic co-analysis tool. To perform symbolic co-analysis, the user provides the application binary and the gate-level netlist to a testbench harness, along with a list of control flow signals to monitor. The testbench instantiates the design, loads the application binary, and provides inputs (Xs) to the application. The testbench also calls the `monitor_x()` system task, providing the user-specified control flow signals as argument. iverilog assimilates all the information into an iverilog-specific intermediate representation (vvp assembly) [20] and starts the simulation. Once the simulation reaches a PC-changing instruction where any of the signals that determine control flow are X, the execution path becomes non-deterministic, and we must explore all possible execution paths. At this point, the simulation is halted, the simulation state is saved, and the Conservative

State Manager (CSM) is alerted. The CSM is a program that maintains a repository of previously-simulated states. A simulation state is indexed by the PC of the PC-changing instruction at which it was observed. When the simulator halts the simulation and provides the simulation state to the CSM, the CSM compares the state with the most conservative state that has been simulated thus far for the same PC. If the current state is a strict subset of the previously-simulated state, this state has already been evaluated, and hence, further simulation is not required. If the current state is not a strict subset, the CSM generates a more conservative state that covers both states by merging the current state and existing conservative state. Once the new conservative state is formed, appropriate control flow signals are set to continue down the possible execution paths from the PC-changing instruction. Algorithm 3 describes the simulation procedure.

The simulation is complete when there are no new states to simulate. We then obtain gate activity information for all explored paths. We combine the activity information to generate the gate activity information for the entire application. The gate activity information indicates all the gates that are exercisable by the application. This information can be used for subsequent application-specific design optimizations. For example, to generate a bespoke processor, unexercisable gates are pruned away and the microprocessor design is re-synthesized to generate a new gate-level netlist with lower area and power consumption. During re-synthesis, fanout values of pruned gates are set to the constant value seen during the symbolic simulation of the target application.

4.3.1 Designing A Testbench For Symbolic Hardware-Software Co-Analysis For Iverilog

Listing 4.1 describes a simple testbench that uses the symbolic simulation feature of the iverilog tool. The user must follow the steps described below to perform symbolic hardware-software co-analysis.

1) The testbench calls two system tasks: `monitor_x()` and `initialization_state()` in an initial block. `monitor_x()` accepts a list of signals that affect control flow as argument, allowing iverilog to halt simulation when the execution path is non-deterministic. `initialization_state()` accepts simulation state as argument to allow iverilog to initialize the processor and simulator states, and begin

Algorithm 3 Symbolic Hardware-Software Co-analysis using iverilog

1. **Procedure** *symbolic_simulation(app_binary, design_netlist, control_signals)*
2. Load the design_netlist and initialize the Memory.
3. Load app_binary into program memory
4. Propagate reset signal
5. $s \leftarrow$ State at start of app_binary
6. $cs \leftarrow$ control_signals
7. Table of previously observed symbolic states, T.insert(s)
8. Stack of un-processed execution paths, U.push(s)
9. $T_p \leftarrow \phi$ // Initialize empty toggle profile
10. $T_n \leftarrow \phi$ // Initialize empty toggle nets
11. **while** U != $\emptyset$ **do**
12. $e \leftarrow U$.pop()
13. e.set_control_signals() // set control signals for a execution path
14.   `$initialize_state(`e`)`
15.   // halt if any of the control signal becomes X
16.   **while** `$monitor_x(`cs`)` == 0 **do**
17. $e' \leftarrow$ propagate_gate_values(e) // simulate this cycle
18. $e \leftarrow e'$ // advance cycle state
19.   **end while**
20. $c \leftarrow T$.get_conservative_state(e)
21.   **if** $e' \not\subset c$ **then**
22. $e'' \leftarrow T$.make_conservative_superstate(c,e')
23. U.push(e'')
24. T_p.save_toggle_profiles(e'')
25.   **else**
26.   break
27.   **end if**
28. **end while**
29. // Merge toggled nets of all the toggled paths.
30. **for all** $p \in T_p$ **do**
31. T_n.append(p)
32. **end for**
33. // Mark driver gates of the corresponding nets as toggled.
34. **for all** $n \in T_n$ **do**
35.   **if** n.toggled() **then**
36. $g \leftarrow n$.getDriverGate()
37. g.setToggled()
38.   **end if**
39. **end for**
40. **for all** $g \in$ design_netlist **do**
41.   **if** g.untoggled **then**
42.   annotate_constant_value(g,s) // record the gate's initial (and final) value
43.   **end if**
44. **end for**

Listing 4.1: Simple verilog test bench harness for starting symbolic simulation

```
initial
begin
    $monitor_x("control_signals.ini");
    $initialize_state("sim_state.log");

    RST_n = 1'b0;
    #100 RST_n = 1'b1;
end

reg [7:0]data_memory[7999:0]; // 8kB data memory

// Instantiate Design.
GateLevelNetList dut(input reg1, reg2,..., data_memory);

initial
begin
    reg1 = 16{1'bx};
    reg2 = 16{1'bx};
    // set input dependent memory locations as X
    for (i = start_loc; i < end_loc; i = i + 1)
    begin
        data_memory = 8'bxxxxxxxx;
    end
end
... // other necessary initializations
```

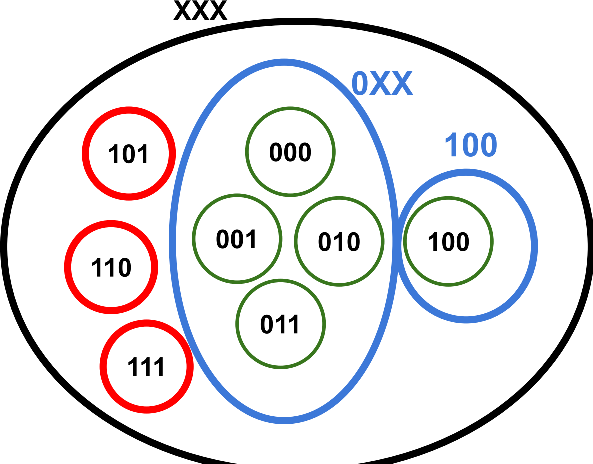
simulation from a previously halted state.

2) The testbench must instantiate and reset the processor.

3) The testbench must initialize the processor inputs – registers and memory – to Xs to allow iverilog to simulate all possible execution paths of the application.

4.3.2 Conservative State Management

Simulation halts if one or more Xs is encountered in a *monitored* state variable or if the simulation terminating condition is met, indicating that all possible application states have been simulated. In case of an X in a monitored signal, we launch multiple instances of iverilog that execute the branches of the simulation where the Xs in the monitored state are re-interpreted as ones or zeros to cover all legal scenarios. Alternatively, we can apply the conservative state optimization proposed in prior works [14]. Using this optimization, a more conservative state of the saved state is generated by merging all the



Simulated states	Unobserved states
000, 001, 010, 011, 100	ϕ
0XX, 100	ϕ
XXX	101, 110, 111

Figure 4.3: Various approaches for conservative state generation exhibit trade-offs between simulation effort and conservative over-approximation. To capture all states in the first row (green) we could either create two conservative states as shown in the second row (blue) or one uber-conservative state as shown in the third row (red).

previously-observed states that match the PC of the current saved state. Applying the conservative state optimization significantly accelerates simulation by allowing many simulation paths that are covered by the conservative state to be discarded.

How conservative states are formed can be configured in the simulator. A designer can choose any approach to form conservative states, depending on convergence and accuracy requirements, as long as the approach ensures that the formed conservative state covers all observed states. For example, the approach used in prior work is to generate a single conservative state by merging simulation states and replacing all differing bits with Xs. Generating a single state to cover all observed states allows the simulation to converge the quickest and is most scalable, but it is also the most conservative, and represents some gates as exercisable that may not actually be exercisable. Consider the in Figure 4.3, where the observed states for a given PC are represented by the green circles. A conservative state of **XXX** encompasses all the observed states, and in addition, covers a few unobserved states. Though this approach reduces simulation time significantly, it can lead to over-approximation of exercisable gates. As another example, consider using a conservative state of **0XX** along with the state **100**, represented by blue circles. This conservative state formulation requires simulation of two

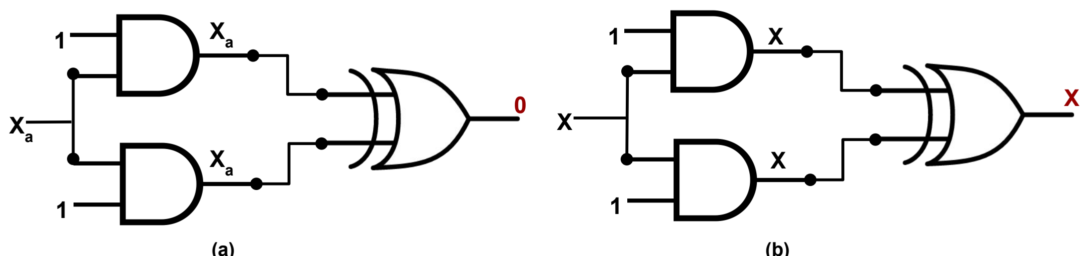


Figure 4.4: Our symbolic tool allows rules for symbol propagation to be customized. The left sub-figure shows a case where circuit inputs are propagated as separate symbolic values, while the right sub-figure shows a case where the symbolic values carry no identifying information and thus cannot be distinguished.

execution paths rather than the original five and avoids representing unobserved states. In our tool, the CSM supports the ability to specify a custom conservative state generation approach by providing the rules of conservative state generation. Another example of a custom approach could be using application constraints to constrain conservative states [17]. The CSM accepts constraints in the form of a text file and uses them to reduce over-approximation of conservative states. The CSM keeps track of all the saved states along with their PC values and generates conservative states to be fed into the next branch in the simulation. CSM is also responsible for triggering the launch of the iverilog instance that simulates the next branch. Since each branch of the simulation can be run by a separate process, launching these processes in parallel can drastically improve simulation time.

4.3.3 Propagation Of Symbols

The simulation tool also allows customization of symbol propagation. Different approaches for propagating Xs are used for different application-specific optimizations. For example, optimizations that require the identification of unexercisable gates must track the propagation of Xs, as this indicates the possibility of a gate being exercised for some application input, while to provide security guarantees, symbols must also propagate taint information [19]. For a less conservative simulation, we may want to track the propagation of each unknown value individually. This can allow simplification when the same symbol recombines at a gate. For example, the left sub-figure of Figure 4.4 shows a case where inputs to the circuit are propagated as separate symbolic values. In

this case, it can be determined that the inputs to the XOR gate have the same unknown value, and the output of the XOR gate is logic 0. In the right sub-figure, no identifying information is propagated with the symbols, so it cannot be determined that the inputs to the XOR gate have the same value, and the output must be assumed to be unknown (X). The latter approach is easier and more scalable to simulate, while the former is less conservative.

4.4 Evaluation

In this section, we demonstrate the generality of the novel analysis tool by evaluating our methodology on three different processor implementations, each based on a different ISA – openMSP430 [25] – an open-source version of one of the most popular ULP processors [33, 34], a custom implementation of an open-source 32-bit MIPS processor – bm32 [35] – and DarkRISCV SoC [36], a RISCV implementation that implements the RV32e ISA [37] with integer registers reduced to 16 bits. Our implementation of DarkRISCV only modeled the processor core and memory. The processor designs are synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [26] and Cadence EDI System [27].

Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using our symbolic simulation tool. Section 6.5.2 lists our benchmark applications. We show results for the benchmarks that fit in the program memory of the processors. Table 4.2 lists the selected processors and their features.

The gate-level simulations were performed using an enhanced version of iverilog [20] written in C++ and a Conservative State Manager written in Perl. The CSM uses the conservative state approach used in prior work [14].

Benchmarks are chosen to be representative of emerging ULP application domains such as wearables, internet of things, and sensor networks [28]. Also, benchmarks were selected to represent a range of complexity in terms of control flow and execution length.

Experiments were performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz frequency, 64GB RAM).

Table 4.1: Benchmark applications

Benchmark	**Description**
Div	Unsigned integer division
inSort	in-place insertion sort
binSearch	Binary search
tHold	Digital threshold detector
mult	unsigned multiplication
tea8	TEA encryption algorithm

Table 4.2: Target platform characterization

Design	ISA	Features
bm32	MIPS32	32-bit MIPS implementation, with hardware multiplier.
openMSP430	MSP430	16bit microcontroller with 16x16 Hardware Multiplier, Watchdog, GPIO, TimerA
dr5	RV32e	32-bit RISCV embedded ISA with 16 integer register, 3 stage pipeline.

Using our tool, we run conservative-state based symbolic simulation for all the applications in Section 6.5.2 on three microprocessor designs – openMSP430 (MSP430), bm32 (MIPS32), and dr5 (RV32e) and generate the input-independent gate activity profile. We then prune away the unused gates and re-synthesize the design to generate an area and energy efficient *bespoke* processor, as in [14].

4.4.1 Validation

To verify that the bespoke netlist generated with our generalized simulation tool works correctly, we simulate the application behavior using fixed known inputs on both the original and the bespoke gate-level netlist. We verified that the outputs from both the designs are the same. We also verified that the set of exercised gates for the fixed input run is a *subset* of the set of exercisable gates reported by our tool. Also, to ensure

Table 4.3: Gate count analysis

Benchmark	BM32 tgc: 16795		omsp430 tgc: 7218		darkriscv tgc: 7578	
	GateCount	% reduction	GateCount	% reduction	GateCount	% reduction
Div	12008	28.5	3175	56.01	6399	15.56
inSort	12210	27.3	3098	57.08	6402	15.52
binSearch	12200	27.36	3115	56.84	6324	16.55
tHold	12139	27.72	2970	58.85	6259	17.41
mult	12707	24.34	3651	49.42	6299	16.88
tea8	12340	26.53	2755	61.83	6577	13.21

that the bespoke optimization enhancements made to iverilog do not affect the existing simulation capabilities, we verified that the event list from the baseline iverilog version matches the iverilog version after our enhancements at simulation points for applications that are picked at random.

4.4.2 Exercisable Gates

Table 4.3 shows the number of gates marked as exercisable by an application for the three designs. The total number of gates in the three microprocessor designs – bm32, openMSP30, dr5 – are 16795, 7218, and 7578, respectively. Using our tool to perform symbolic hardware-software co-analysis, we achieve a gate count reduction of 27%, 56% and 16% for these processors, respectively. Figure 4.5 shows the percentage reduction of the toggled gates for all benchmarks in Section 6.5.2. We observe that designs with external peripherals tend to have a higher gate count reduction. This is because, for applications that do not use peripherals, the set of gates representing the peripheral logic will not be exercised and can be safely removed. Since dr5 does not contain any peripheral logic such as a multiplier, it exhibits a relatively smaller reduction in the toggled gate count.

4.4.3 Simulation paths

From the simulation paths reported in Figure 4.6, we observe that bm32 and dr5 require significantly more simulation paths than openMSP430 to complete symbolic simulation. This is because of a fundamental difference in how `compare` instructions are

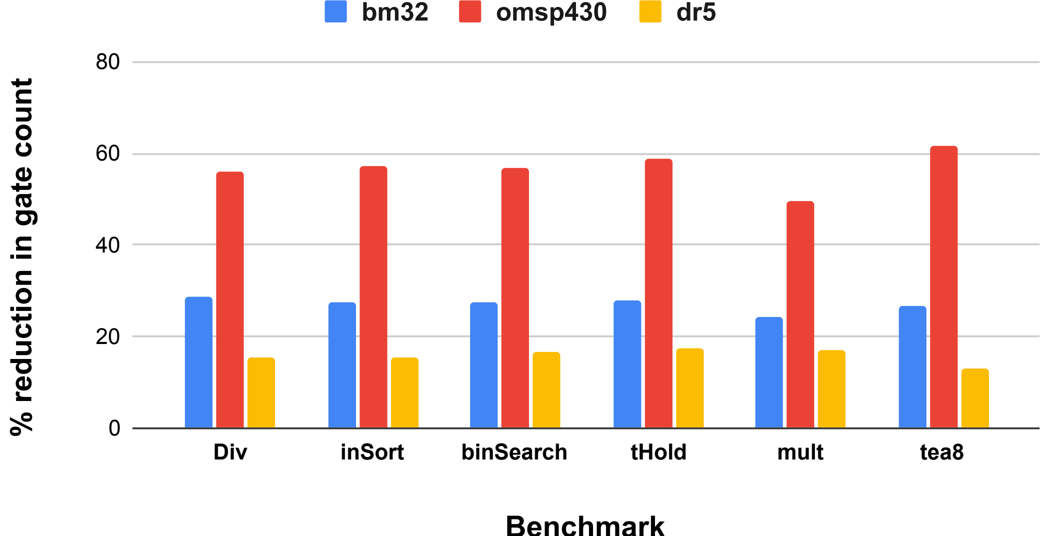


Figure 4.5: Benchmarks run on MSP430 processor have a higher reduction in exercisable gate count compared to MIPS and RISCV processors because of the presence of unused peripherals in MSP430.

implemented in the designs and how that affects conditional jumps in an application. In openMSP430, the result of the `compare` instruction is stored in program status word in the form of N, Z, C, and V flags. Based on the value of these flags (1 or 0), conditional jumps are resolved. In bm32 and dr5, on the other hand, the `compare` instruction is implemented as a subtraction operation, and the resulting value is stored in a 16-bit register, which is used to resolve conditional jumps. As discussed in Section 4.3 we halt the simulation when the output of a `compare` instruction preceding a conditional jump resolves to one or more Xs. In the case of openMSP430, this means when any of the NZCV flags of the status register is an X. In the case of bm32 and dr5, this means that the 16-bit register that holds the result of subtraction contains one or more Xs. If the 16-bit result register already contains an X, subsequent subtractions (such as `compare` used to evaluate loop termination conditions) would increase the number of Xs in the register. In most applications, all possible execution paths are only evaluated when the entire register fills with Xs. This significantly increases the number of paths that need to be evaluated for bm32 and dr5 processors. Since the NZCV flags in openMSP430 are 1-bit each, there are no additional Xs incurred at every `compare` instruction. This

Table 4.4: Simulation path and runtime analysis

Benchmark	BM32 tgc: 16795			omsp430 tgc: 7218			darkriscv tgc: 7578		
	paths created	skipped	simulated cycles	paths created	skipped	simulated cycles	paths created	skipped	simulated cycles
Div	327	112	53202	17	8	776	325	112	13149
inSort	315	130	35044	230	118	18086	319	132	9382
binSearch	941	190	154198	119	62	9715	829	190	2374
tHold	191	68	17168	293	184	13030	191	68	4690
mult	1	0	528	1	0	258	175	60	5790
tea8	1	0	10018	1	0	3852	1	0	4534

means that openMSP430 is able to converge faster, while for bm32 and dr5, several simulation instances are necessary to reach a simulation state that represents all possible subtraction operations. Due to the use of status bits (NZCV flags), benchmarks compiled for openMSP430 also have fewer conditional branch instructions compared to benchmarks in other processors, leading to fewer explored paths.

Another factor that significantly affected the simulation time for dr5 is the lack of a hardware multiplier module. As such, the compiler for dr5 performs multiplication in software using a library implementation of multiplication in the form of repeated additions in a loop. This leads to the use of input-dependent conditional branches to perform multiplication in dr5. Since input-dependent conditional branches lead to the generation of multiple simulation paths, we see that for the benchmark mult, dr5 has more than one simulation path in Figure 4.6, while the number of simulation paths for the other two processors that use a hardware multiplier is one.

Finally, Figure 4.6 shows that for the benchmark tHold, the number of simulated paths is higher for openMSP430 compared to bm32 and dr5, contradicting the trend seen in the other benchmarks. This is because the compiled binary for openMSP430 had three conditional branch instructions vs two in dr5 and bm32. Hence, in openMSP430, the number of execution split points in each loop iteration of tHold is three, compared to only two for dr5 and bm32. This difference quickly adds up as the symbolic execution tree is built, leading to a higher number of simulation paths for openMSP430. Table 4.4 provides the number of simulation paths created and skipped, along with the simulated cycles for each application in all the designs.

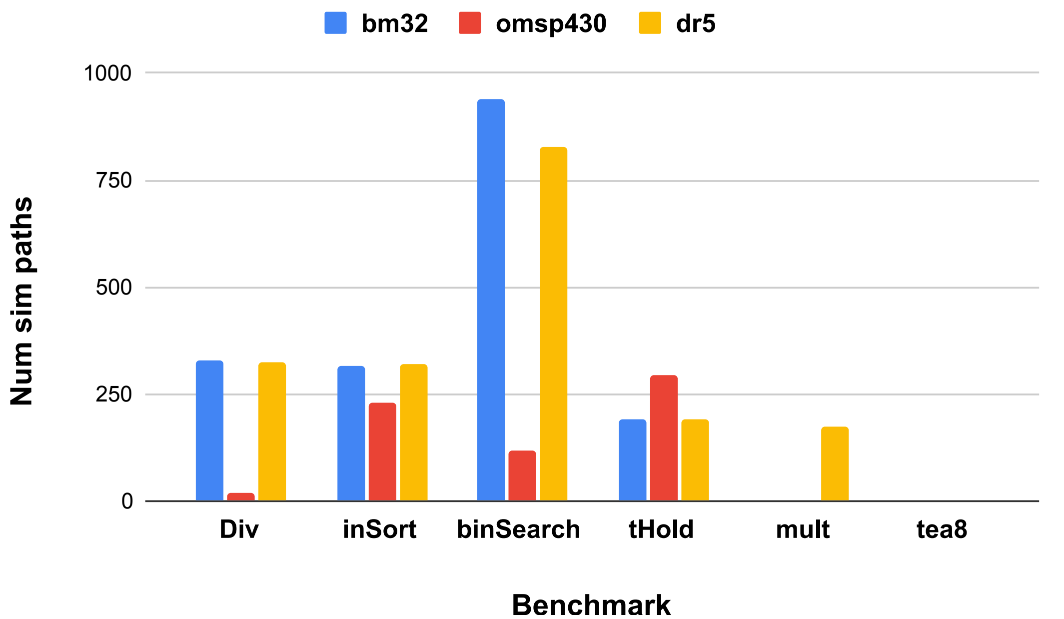


Figure 4.6: Benchmarks run on MIPS and RISCV processors have a higher number of simulated paths because a 16-bit register is used to indicate branch conditions, whereas in MSP430, a 1-bit register is used, resulting in fewer conservative states.

4.5 Related Work

Prior works on application-specific system design and optimization propose symbolic hardware-software co-analysis and demonstrate its use in a number of applications, from providing security guarantees in embedded systems [19], to performing application-specific optimizations that reduce power and energy without sacrificing performance or functionality [12, 13, 16, 38], to automatically generating application-specific bespoke processors for ultra-low-power embedded systems [14]. However, prior works rely on developing a custom simulator for each processor to be analyzed and optimized. Since this is a challenging and time-consuming endeavor that is not scalable, prior works only demonstrated results for a single processor (openMSP430). In our work, we develop a design-agnostic symbolic simulation tool that can apply symbolic hardware-software co-analysis techniques to any digital design and application. Our tool offers a scalable approach to easily extend symbolic analysis and subsequently enable application-specific optimization for new designs.

Prior work on property-driven automatic hardware transformation [39] developed a property-driven framework for automatically generating hardware for a reduced ISA, where a specified list of instructions or ISA features are not supported. The work uses a property library to annotate all gates in the design and performs property checking to identify gates for which the properties are verified. Developing a property library that encodes ISA restrictions for each application is a manual process that can be both challenging and time-consuming. Our symbolic simulation tool, on the other hand, can easily analyze a new design with minimal user effort or expertise. Further, our tool is able to handle designs in any format – RTL or gate-level netlist – described in any hardware description language, e.g., verilog, VHDL, or system verilog.

In our work, we discuss saving and restoring simulation state in iverilog. Restoring simulation state involves assigning values to design elements, such as nets and registers. Prior works have used verilog constructs such as `force` and `release` for fault injection in design elements [32]. However, at any simulation point, `force` and `release` allow us to assign only one value to a design element. To assign a different value, the testbench must be modified and recompiled. Also, the simulation must be restarted from the beginning. By saving and restoring simulation states, we avoid this overhead. Using `force` and `release`, we cannot split the simulation and launch multiple instances. Our approach allows us to parallelize simulations for different execution paths.

4.6 Summary

Current state-of-the-art symbolic simulation tools for hardware-software co-analysis are restricted in their applicability, since prior work relies on a costly process of building a custom simulation tool for each processor design. In this chapter, we described how we modified iverilog to support propagation of symbolic values, conservative state generation and simulation, monitoring of critical control signals, and saving and restoration of simulation states, thus creating a design-agnostic symbolic simulation tool for hardware-software co-analysis. We demonstrated the generality of our tool by performing symbolic analysis on three embedded processors with different ISAs, and we also used analysis results from our tool to generate bespoke processors for each processor design and discussed the impact of architectures on the results and simulation times.

Our results demonstrate the versatility of our simulation tool and the uniqueness of each design with respect to symbolic analysis and the bespoke methodology.

Chapter 5

Application-Specific Architecture Selection

In the last chapter, we introduced a symbolic simulation tool that performs hardware-software co-analysis on any processor-application pair. The versatility of our tool opens the possibility of a wide scope of research and analysis. By facilitating symbolic simulation of an application on a processor netlist, our tool has simplified characterizing gate-level behavior of an application. By pruning away gates in a processor that the system's target application is guaranteed not to use, significant energy savings can be achieved. The resulting *bespoke* processor is a pruned-down version of the original and the processor logic is unchanged from the perspective of the application, the benefits of using a GPP remain largely intact.

Application-specific design-level optimizations are effective when optimizing a processor. However, if the goal is to design a application-specific processor that is optimal in terms of a metric such as energy efficiency, processor architecture must also be optimized. But, architectural parameter space is huge considering the choices for design implementation, list of possible peripherals, size of register-memory space, and many more. Synthesizing a new design for each parameter variation and performing application-specific optimizations is not feasible due to the large architecture parameter space. Moreover, application-specific optimizations impacts a processor in a non-linear fashion, i.e., the

impact of an optimization is different for a different processor-application pair, superposition cannot be applied. Selecting the optimal processor architecture for an application and applying application-specific optimization does not generate the optimal bespoke processor for the application. Given the wide usage of Machine Learning (ML) for effective design space exploration, we sought the aid of ML to efficiently explore the architectural parameter space and predict the quality of a processor architecture choice for a target application using features extracted from processor design choices and characteristics of the application. In this chapter, we demonstrate the use of ML model in navigating through the architecture parameter space. We also show how the predictions from the ML model help improve the area and power savings by choosing the optimal architecture for a bespoke processor for a target application.

5.1 Effect Of Bespoke Process On Architectural Variants

As discussed in previous chapters, a characteristic of many embedded devices is that they run a single application over and over throughout their lifetime. This opened up the door for application-specific optimizations to improve processor efficiency. Using bespoke processor, we tailor a processor for a target application and reduce power and area without sacrificing application performance or functionality. However, there are a wide variety of general purpose embedded processors with different characteristics such as ISA, memory size, and microarchitectural features, and for a single application, each different processor can be used to generate a bespoke processor with different efficiency for a system designer's metric of interest (power, area, performance, energy, etc.). Some processor architectures naturally lend themselves to greater optimization for a specific application, and it is not clear how to choose the architecture that is best suited for a bespoke processor's target application and efficiency metric.

Figure 5.1 shows the gate count for several architectural variants of the darkriscv processor [36]. Each architectural variant was generated by selecting different architectures for the adders and multipliers used in the processor. The orange data points represent the total number of gates in the placed and routed design for each architectural variant. The points are sorted so that the gate count increases from left to right.

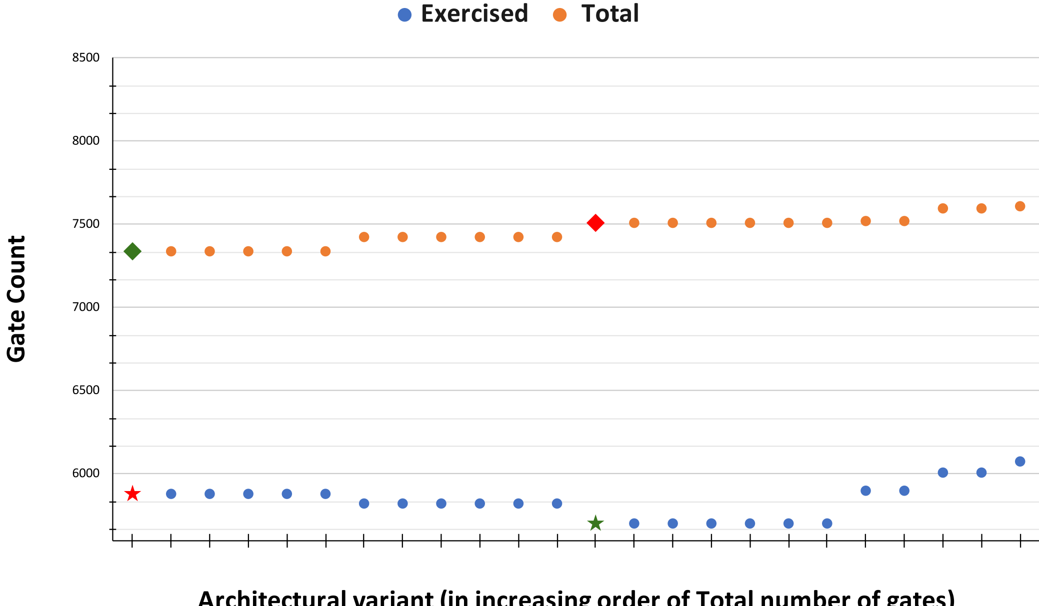


Figure 5.1: This plot shows the total gate count for various architectural variants of the darkriscv processor both before and after bespoke customization for the tea8 application. The optimal processor variant before customization (green diamond) is different than the optimal bespoke processor variant after customization (green star).

The blue data points correspond to the number of gates in a bespoke processor generated from each architectural variant. The bespoke processors are tailored for tea8 – a popular encryption algorithm used in embedded systems [40] – using the conservative state based symbolic hardware-software co-analysis technique.

In Figure 5.1, the leftmost architectural variant (marked with a green diamond) has the lowest gate count. This design used an AND-based non-Booth multiplier and a conditional sum adder [41]. However, after eliminating unexercisable logic for tea8 in each architectural variant, the leftmost variant does not correspond to the bespoke processor with the lowest gate count. The bespoke design with the lowest gate count (indicated by the green star) uses a Booth-encoded radix-8 multiplier and a carry-save adder. Thus, for this example, selecting the optimal architectural variant on which to perform bespoke customization leads to a suboptimal design after customization.

Furthermore, this example, which only considers changes to the adder and multiplier

architectures for the processor, required a significant amount of simulation and design automation effort to generate and evaluate all the design variants. As the architectural parameter space expands to include more architectural options, the time required to enumerate and evaluate all options in order to identify the optimal architectural variant quickly becomes prohibitive.

5.2 Effect Of Architectural Variants On Efficiency Metric

In the last section, we discussed the effect of bespoke process on architectural variants, specifically adder and multiplier implementations. We showed that the impact of bespoke process on architecture is non-deterministic. In this section, we show how architecture impacts the efficiency metric.

5.2.1 Processor Architectures

Figure 5.2 shows the energy per bit and NAND-equivalent area per bit for different bespoke processors tailored for applications based on multiplication (mult) and binary search (binsearch). In this example, the architectural variants correspond to different processor architectures – MSP430-based openMSP430 [25], MIPS-based bm32 [35], and RISC-V-based darkriscv [36].[1] Energy per bit is computed by dividing the energy required to execute the application by the bit-width of the processor. Area per bit is computed analogously. Per-bit efficiency metrics are used because the processors have different bit widths; openMSP430 is a 16-bit processor while bm32 and darkriscv are 32-bit processors. We observe that in terms of energy per bit, openMPS430 is a better choice for the mult application while bm32 is a better choice for binSearch. Both openMSP430 and bm32 have lower energy per bit than darkriscv for the mult application due to the presence of hardware multipliers in those architectures. Although the bespoke processors generated from the darkriscv architecture have higher energy per bit than those generated from the other two architectures, the darkriscv-based bespoke processors have the lowest area per bit for both applications. The results in Figure 5.2 demonstrate that the best processor architecture from which to generate a bespoke

[1] All processor architectural variants use a Booth-encoded radix-4 multiplier architecture and a Sklansky adder architecture.

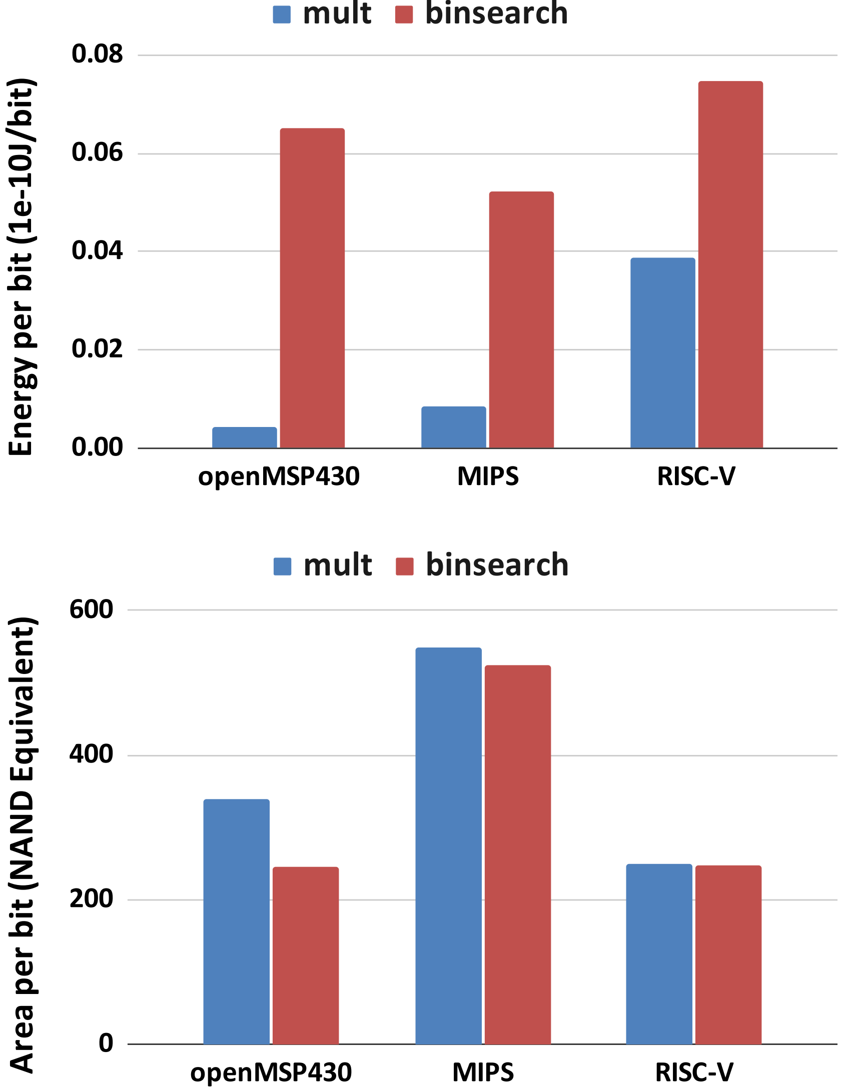


Figure 5.2: The plots compare per-bit energy consumption (top) and area (bottom) for bespoke processors tailored for mult and binsearch applications, starting from three distinct processor architectures – MSP430, MIPS, and RISC-V. The MSP430-based bespoke processor has the lowest per-bit energy consumption for the mult application, but the MIPS-based design has the lowest energy for the binsearch application. On the other hand, the RISC-V-based design has the lowest per-bit area for each application. The results demonstrate that the best processor architecture from which to generate a bespoke processor differs based on the target application and efficiency metric.

processor differs based on the target application and efficiency metric.

5.2.2 Hardware Accelerators

In this section, we present results for bespoke hardware accelerators generated from a 32-bit Discrete Cosine Transform (DCT) accelerator. We consider folded and un-folded architectural variants of the accelerator, and we perform bespoke customization for various applications in which the required bit precision of the input signal is varied from 4 bits to 32 bits.[2] We present the energy and area of the resulting bespoke accelerators in Figure 5.3. Note that we do not compare per-bit efficiency metrics in this case, since the different bit widths correspond to different applications for which the accelerator is customized, not different architectural variants. The figure shows that the folded architecture is more area-efficient independent of input bit width. This is not surprising, since the un-folded design is essentially "parallelized" so that it can handle multiple inputs at the same time.

Comparing the energy of the bespoke designs generated from the two architectural variants, the un-folded architecture results the lower energy for an input bit width of 32, while the folded architecture results in lower energy for lower bit widths. At 32-bit precision, the un-folded design, which generates the output faster than the folded design, consumes less energy. However, since the area reductions for bespoke customization are not significant as bit width reduces, the significantly lower area of the folded architecture outweighs the time savings of the un-folded design at lower input bit widths. As in the previous example, the best accelerator architecture from which to generate a bespoke accelerator differs based on the target application and efficiency metric.

5.3 Motivation

The above discussions leads us to conclude that for a given target application and efficiency metric, the architecture from which a bespoke processor or accelerator is generated can have a significant impact on efficiency for a system designer's metric of choice. In addition to the differences between the un-tailored architectures themselves, the significantly different and non-uniform impact of bespoke customization on different

[2] Both filter architectures use Booth-encoded radix-8 multipliers and Sklansky adders.

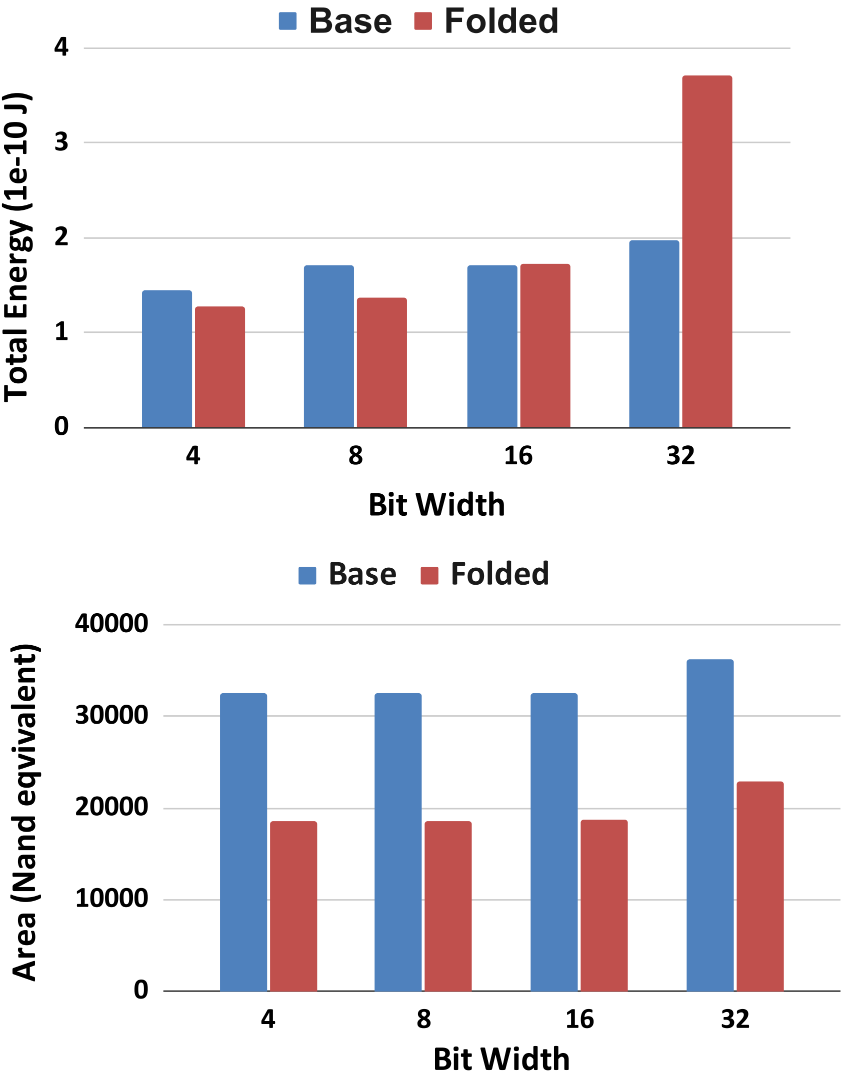


Figure 5.3: The plots compare energy consumption (top) and area (bottom) for bespoke accelerators tailored for applications with different input bit precision, starting from folded and un-folded architectural variants of a 32-bit DCT filter DSP accelerator. The x-axis represents input signal bit width, corresponding to different applications that require different levels of precision. Bespoke accelerators generated from the folded architecture have lower area for all input bit widths, but for a bit width of 32, the un-folded accelerator has lower energy due to its lower computation time. The best accelerator architecture from which to generate a bespoke accelerator differs based on the target application (bit width) and efficiency metric.

hardware architectures leads to a large and rich design space to be explored in order to identify the most efficient bespoke design for an application. Given any non-trivial architectural parameter space, a brute force approach for exploring this search space will be prohibitively expensive due to the runtime complexity of symbolically evaluating the application on the hardware to evaluate all possible executions of the application and subsequently applying design automation techniques to perform bespoke customization, synthesis, placement, and routing of the design, and metric evaluation on the placed and routed design. This motivates us to explore intelligent and efficient means of identifying the architectural variant from which to generate a bespoke design for an application such that efficiency is optimized for a metric of choice. In the next section, we describe the development of a machine learning model that can predict the value of a chosen efficiency metric for a bespoke design that is tailored for a target application. This model can be used to significantly speed up architecture selection for a bespoke design.

5.4 Application-Specific Architecture Selection

Selecting the starting architecture from which to generate a bespoke processor for a given target application and efficiency metric can be a computationally expensive task. As explained in Section 5.3, the computational overhead arises from the fact that even a relatively small architectural parameter space can result in numerous architectural variants, and generation of a bespoke processor from each variant requires a symbolic simulation on the gate-level netlist of the processor that explores all possible execution paths of the target application, plus running electronic design automation tools to perform design pruning and layout. For example, suppose the system designer wants to explore an architectural parameter space with five processor architectures, ten different adder architectures, and five different multiplier architectures. Even for this relatively small parameterization, the resulting design space would contain 250 architectural variants. Even minor expansion of the architectural parameter space to include other microarchitectural parameters (e.g., divider architectures, floating point arithmetic units, pipeline stage implementations, hazard avoidance mechanisms, etc.) can quickly cause the number of architectural variants to explode. To identify the most efficient bespoke design for an application, a designer would have to perform all the steps to generate

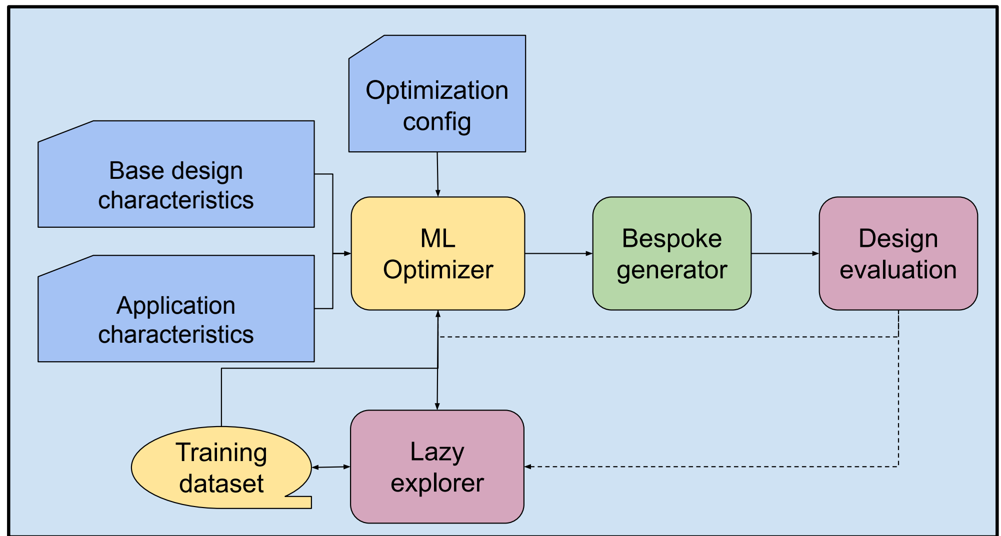


Figure 5.4: Our machine learning model for selecting an architectural configuration from which to generate a bespoke processor uses architecture features of the baseline design and application characteristics to predict metric values for each architectural configuration in the architectural parameter space. A short-list of candidate architectures is evaluated more thoroughly to identify the most efficient architectural variant.

a bespoke processor from each variant and subsequently evaluate the efficiency metric of choice on each bespoke design, making application-specific design space exploration prohibitively expensive in most cases.

Rather than enumerating, generating, and evaluating a bespoke design corresponding to each architectural variant, our approach for identifying an efficient architecture from which to generate a bespoke processor, outlined in Figure 5.4, uses a machine learning model (ML-optimizer) that can quickly predict the metric of interest for a given application-design pair. We extract features from the application and the base processor design (see Section 5.4.1) and provide them as inputs to the machine learning model, along with the architectural configuration (e.g., the processor architecture and type of adder and multiplier architectures) for which a prediction is desired. Note that all the inputs to the model can be determined without requiring a simulation or synthesis campaign. For example, the training of the ML model and the metric value prediction for all possible DSP configurations took an average of 30 seconds for each

metric. We then selectively run the expensive process of synthesizing the design, performing symbolic simulation of the application, generating the bespoke processor, and evaluating the metric on the pruned design for a limited number of candidate architectures identified by the model. For example, we only run the bespoke flow on the top 10% of predicted designs. We then annotate the features of the evaluated designs with the true metric that was generated and add the metric-annotated feature vector to the training set. By storing the true features of the evaluated metrics in our training set, we can train our model online.

Adding to the training set is performed using a lazy exploration algorithm which only updates the training set if both the base design and the application for which inference was performed are not in the training set. In the case where at least one of base design and application are already seen in past training, we do not add new data points to the training set.

5.4.1 Feature Extraction

Our machine learning model predicts the value of an efficiency metric for a bespoke design that is generated from a starting architectural configuration using features extracted from two sources – the application and the baseline architecture.

Application Characteristics: Application features capture the microarchitecture-agnostic characteristics of the application. We extract application features after the application is compiled to a binary. Extracted features can include the bit-width of the data that the application processes, the size of the application, and the mix of instructions in the application.

Baseline Architecture Characteristics: These features capture the architecture characteristics of the processor. To efficiently extract the baseline architecture features, we synthesize the baseline architecture once and extract features from the synthesis report, such as number of adders, number of multipliers, number of registers, number of register-to-register paths, and average register-to-register path length in number of gates. This significantly reduces the need for manually reading the design and extracting design features. We also use pipeline depth as a feature, which requires designer input to specify the pipeline depth.

Architectural Configuration The architectural configuration refers to values of architectural parameters (e.g., architectures for various arithmetic units) for which a metric prediction is desired. We use a one-hot encoded string to capture the type of functional unit used in a particular optimization configuration. In our experiments, we explore a total of 58 different optimization configurations.

5.4.2 Model Selection

We evaluated several machine learning models for design space exploration; three relevant candidates are explained below.

- **Linear Regression with Lasso:** Since the goal of the model is to predict the value of a metric, the problem can be framed as a regression problem. We trained a linear regression model with lasso using several features that we extracted from the application, processor architecture, and optimization configuration. This model performed poorly and was too simple to predict the metric value corresponding to an input feature vector.

- **Regression Trees:** Since a linear regression model did not perform well, we could infer that the features interact non-linearly to predict the metric. This led us to use a regression tree-based machine learning model to predict the metric. While the model did capture the impact of certain features on the metric to be predicted, it did not scale well with the number of features that we wanted to train the model on.

- **Neural Networks:** Finally, we developed a neural network-based model that not only scaled well with the number of features but also predicted different metrics accurately. We used a four layer neural network for each metric to be predicted, where the final layer contained only a single value. The configuration of our neural network is presented in Figure 5.5. The activation functions are not shown because we chose different activation functions for different metrics. For predicting energy, each layer had a tanh activation, and for predicting area, each layer had a ReLU activation.

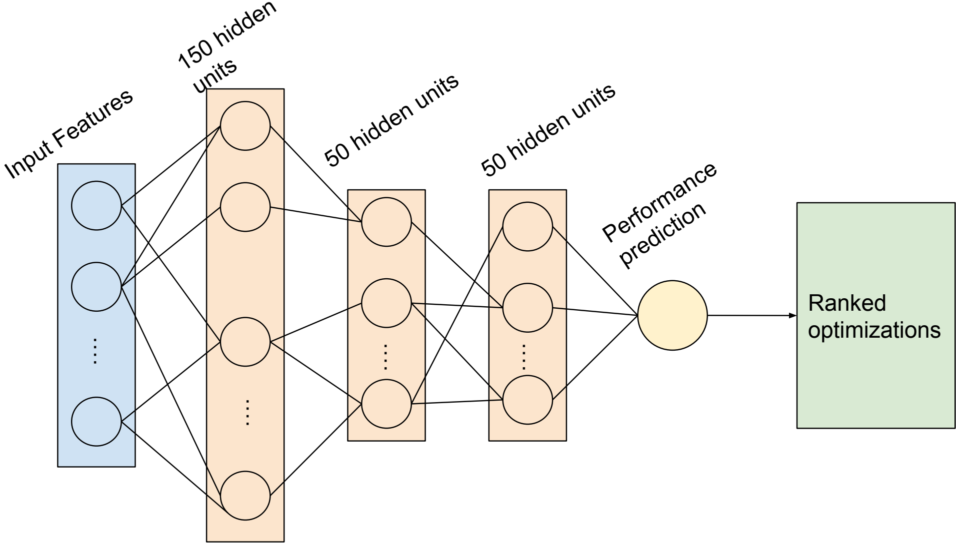


Figure 5.5: We use a neural network that predicts different desired metrics. There is a slight variation in the models that predict area and energy metrics. The model that predicts energy uses the tanh activation function, while the model that predicts area uses ReLU activation.

5.4.3 Training The Model

We employ a leave one out strategy to train our model, where we leave one data point from our training set and train our model using the rest of the data points. We then test our model for accuracy using the data point that was left out.

5.4.4 Prediction, Ranking, And Architecture Selection

We train our model to predict a metric value for the bespoke processors for a target application generated from all possible architectural configurations. While our machine learning models have high prediction accuracy, they do not always have perfect accuracy. As such, it is possible that the predicted rank of each bespoke design may not perfectly align with ground truth, i.e., the rank obtained by exploring the entire design space by synthesizing, simulating, pruning, and evaluating. In other words, the true optimal design might not have a predicted rank of 1. However, for a model with high accuracy, the predicted rank of the true optimal design should be close to 1. We exploit this fact

by running several of the top-ranked designs through the full evaluation flow. In our experiments, the top 10% predicted designs always contained the true optimal design (see Section 5.5).

5.4.5 Application-Specific Architecture Selection For DSP Circuits

Several embedded processors use DSP accelerators to reduce energy consumption and augment the processor's performance for key computational kernels. Each DSP accelerator can have multiple architectural variants. For example, a filter can be pipelined or folded and can use a different number of taps or pipeline stages. Depending on the requirements of the target software application, a DSP circuit can be analyzed and pruned to create a bespoke accelerator with reduced power and area. This can be accomplished by performing an input-independent simulation for the target application on the DSP circuit and pruning away logic that is not exercised during the simulation.

Since each DSP accelerator can have several architectural parameters, including a variety of choices for arithmetic unit architectures, there exists a large design space to be explored to identify the optimal architectural variant from which to generate a bespoke architecture. To explore this design space, we use the methodology outlined in Figure 5.4 to train a neural network model for different DSP accelerators. Some design features, such as pipeline depth, number of adders, number of multipliers, and input width are even more relevant for DSP accelerators than general purpose processors.

5.5 Evaluation

In this section, we evaluate the accuracy of our machine learning model in predicting the rank of the optimal architectural variant for a particular application. For the evaluations on general purpose processors, we used three designs listed in Table 5.1: openMSP430 [25] – an open-source version of the popular ultra-low-power processor MSP430, bm32 [35] – a custom implementation of an open-source 32-bit MIPS processor, and DarkRISCV SoC [36] – a RISCV processor that implements the RV32e ISA [37] with integer registers reduced to 16 bits. We also performed evaluations on ten DSP accelerator designs listed in Table 5.2: FIR Filter (pipelined, folded and base), IIR

Table 5.1: General purpose processors evaluated

Processor	**ISA**	**Features**
bm32	MIPS32	32-bit MIPS implementation with hardware multiplier
openMSP430	MSP430	16-bit microcontroller with 16x16 Hardware Multiplier, Watchdog, GPIO, TimerA
darcriscv	RV32e	32-bit RISCV embedded ISA with 16 integer registers, 3-stage pipeline

Filter, DCT (folded and base), Butterfly, L1 norm, L2 norm, and Sobel. DSP accelerators were chosen from prior work on noise-tolerant accelerator design [42, 43]. All designs were synthesized using TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [26]. For each design, we explored an architectural parameter space with 58 different architectural variants. The variants are produced by using the Synopsys DesignWare Library [41] to select different adder and multiplier architectures to be instantiated in each processor or accelerator design. Table 5.3 distinguishes the architectural variants evaluated for each general purpose processor and DSP accelerator architecture. We included seven adder architectures and nine multiplier architectures, leading to a search space of 56 possible combinations. We also used Design Compiler's `area` and `speed` optimizations, which override any adder or multiplier architecture choice. This leads to an architectural parameter space of 58 architectural variants for each architecture.

Section 6.5.2 describes the embedded benchmarks used to evaluate the general purpose embedded processor designs. Benchmarks are chosen to be representative of emerging ULP application domains such as wearables, internet of things, and sensor networks [28]. Also, benchmarks were selected to represent a range of complexity in terms of control flow and execution length. To generate a bespoke design for a target application, we identified unexercisable logic that can be pruned from a design by performing an input-independent simulation of the application on the synthesized gate-level netlist

Table 5.2: DSP accelerators evaluated

DSP Accelerator	Architectural Variants	Notes
FIR	pipelined, folded, base	32-bit 4-tap FIR filter
IIR	base	32-bit 4-tap IIR filter
DCT	folded, base	32-bit Discrete Cosine Transform
Butterfly	base	32-bit Butterfly circuit
L1	base	32-bit L1 norm computation circuit
L2	base	32-bit L2 norm computation circuit
Sobel	base	Sobel Filter circuit

of the design using the tool described in Chapter 4 [21]. For model development, we used machine learning models that are available in the Scikit-Learn module available in Python [44].

We present our evaluation in two parts. First, we evaluate our methodology and model for general purpose processors described in Table 5.1 using the applications described in Section 6.5.2. We then evaluate our methodology and model for DSP accelerator circuits described in Table 5.2 with application input signals of varying precision. For all evaluations, we generate bespoke designs using the architectural variants described in Table 5.3.

5.5.1 Design Space Exploration For Bespoke General Purpose Processors

In this section, we evaluate our neural network architecture selection model for three general purpose processors with different ISAs and microarchitectures. Since the optimizations we applied are combinational, the sequential behavior of the processor microarchitecture is unaffected. This information is captured by the register-to-register connectivity of a processor, which can be extracted from the processor's synthesized gate-level netlist. We synthesized the baseline architecture of the processor without any explicit optimizations.We then extracted architectural features such as the number of

Table 5.3: Architectural variants explored

ALU type	Architectural variant	Description
Adder	`ling_adder`	Ling Adder
	`hybrid_adder`	Hybrid Adder
	`carry_select_adder_cell`	Carry Select Adder
	`cond_sum_adder`	Conditional Sum Adder
	`sklansky_adder`	Sklansky Adder
	`brent_kung_adder`	Brent Kung Adder
	`bounded_fanout_adder`	Bounded Fanout Adder
Multiplier	`and`	AND-based non-booth encoded multiplier
	`nand`	NAND-based non-booth encoded multiplier
	`and_radix4`	AND-based non-booth encoded radix 4 multiplier
	`nand_radix4`	NAND-based non-booth encoded radix 4 multiplier
	`benc_radix4`	booth encoded radix-4 multiplier
	`benc_radix8`	booth encoded radix-8 multiplier
	`benc_radix4_mux`	MUX-based booth encoded radix-4 multiplier
	`benc_radix8_mux`	MUX-based booth encoded radix-8 multiplier
-	`area`	Pick the adder and multiplier architectures that minimize area
-	`speed`	Pick the adder and multiplier architectures that minimize delay

Table 5.4: Benchmark applications for general purpose processors

Benchmark	**Description**
binSearch	Binary search
div	Unsigned integer division
inSort	in-place insertion sort
intFilt	integer Filter
mult	unsigned multiplication
tea8	TEA encryption algorithm
tHold	Digital threshold detector

registers in the design, the number of bussed registers (registers that have more than one bit) in the design, number of register-to-register paths (flop-to-flop paths) in the design, and the average length (in gates) of register-to-register paths. Along with register-to-register paths, we also extracted the number of port-to-port paths and average length (in gates) of port-to-port paths. The application features we capture include the size of the application binary and the width of the input. Along with these features, we also specify the architectural configuration as a one-hot encoded vector. We used the leave-one-out strategy to evaluate our model.

Figure 5.6 presents normalized per-bit energy and area for bespoke processors generated for the mult application. The architectural parameter space spans the three GPP architectures in Table 5.1 and all architectural variants described in Table 5.3. Blue triangles indicate actual metric values, and red circles show the corresponding predictions from our model. The energy data in the top sub-figure show that while the absolute prediction accuracy is low for individual metric values (mostly due to a significant offset between actual and predicted values for MIPS), the predictions follow the rank ordering of the actual data. This means that if a certain architectural configuration produces a bespoke processor with a better metric value than another architectural configuration, then the predicted metric values of the two configurations generally follow the same ranking as well. Viewed another way, the slope of the lines that fit the red dots and the blue triangles follow the same trend. We observe a similar trend in the bottom sub-figure, which presents normalized area per bit for the bespoke processors. Note that even though the actual and predicted trend lines would cross for the variants of the MIPS processor, they still maintain roughly the same ordering.

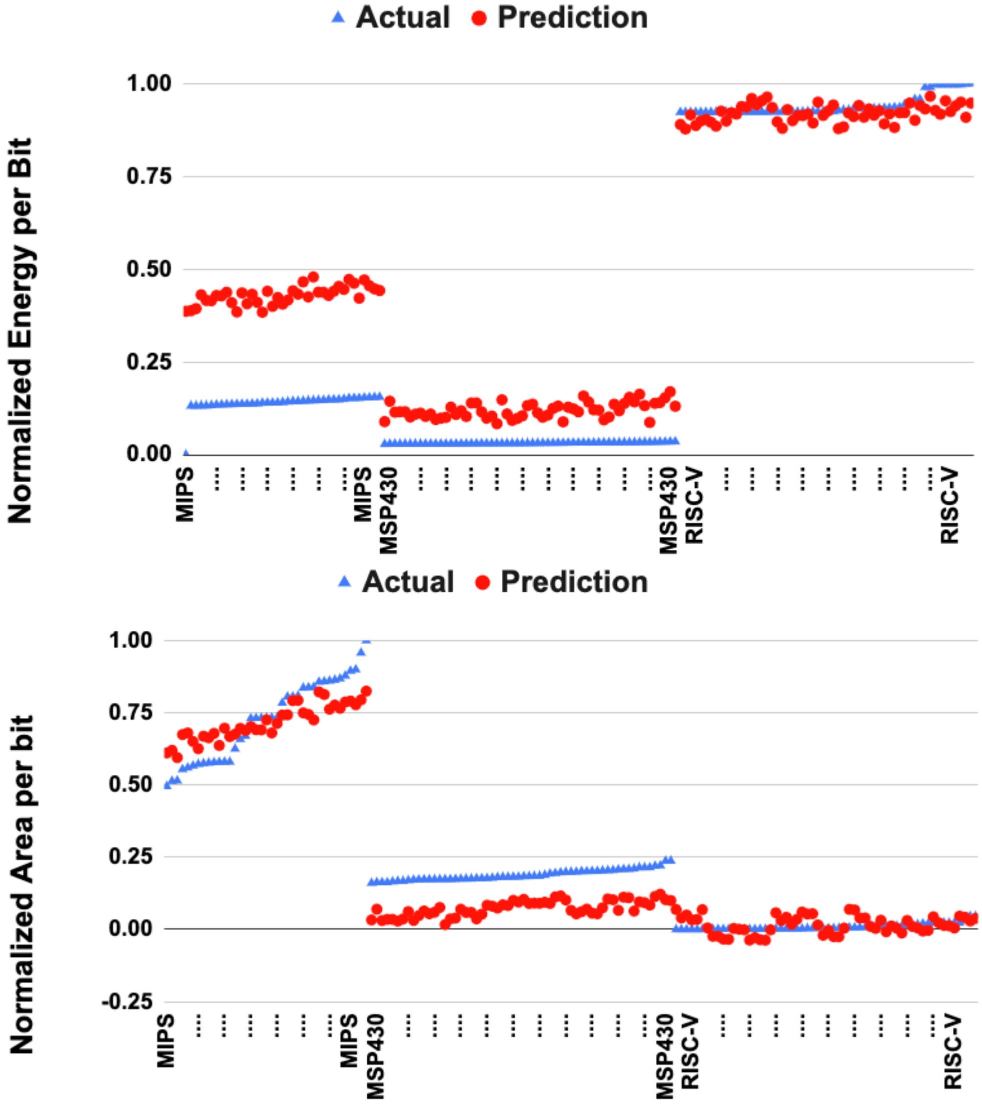


Figure 5.6: This plot shows the normalized energy per bit (top) and normalized area per bit (bottom) predictions of bespoke processors for mult. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend.

Application	**Optimal architecture**	**Optimal Adder and Multiplier**	**Predicted Rank** out of 174
binSearch	openMSP430	carry_select_adder_cell benc_radix4_mux	3
div	openMSP430	carry_select_adder_cell benc_radix4_mux	3
inSort	darkriscv	bounded_fanout_adder nand	7
intFilt	bm32	cond_sum_adder benc_radix8_mux	3
mult	openMSP430	sklansky_adder benc_radix4	11
tea8	bm32	cond_sum_adder benc_radix8_mux	5
tHold	openMSP430	ling_adder benc_radix8_mux	3

Table 5.5: Summary of optimal architecture in terms of area; the model predicts with 100% accuracy in top 10 predictions

With our model, we aim to discover the optimal architecture within the top 10% of architectural candidates predicted by the model. This limits search time by capping the number of full evaluations we perform for architectural configurations. Table 5.5 shows, for each of the benchmark applications, the rank predicted by our model for the architectural variant that minimizes area. The predicted rank of the optimal architecture is always in the top 10% and is usually also in the top 5% or even higher. Also, while the predicted optimal architecture does not always correspond to the actual optimal, the actual metric values for the predicted and actual optimal architectures are very close.

Table 5.6 shows similar results to Table 5.5 for energy instead of area, from which similar conclusions can be drawn. The results in Table 5.5 and Table 5.6 also present the architectural configuration that resulted in the best metric for each application. These results confirm our observations in Section 5.2 that for each application, the best architectural configuration can be different.

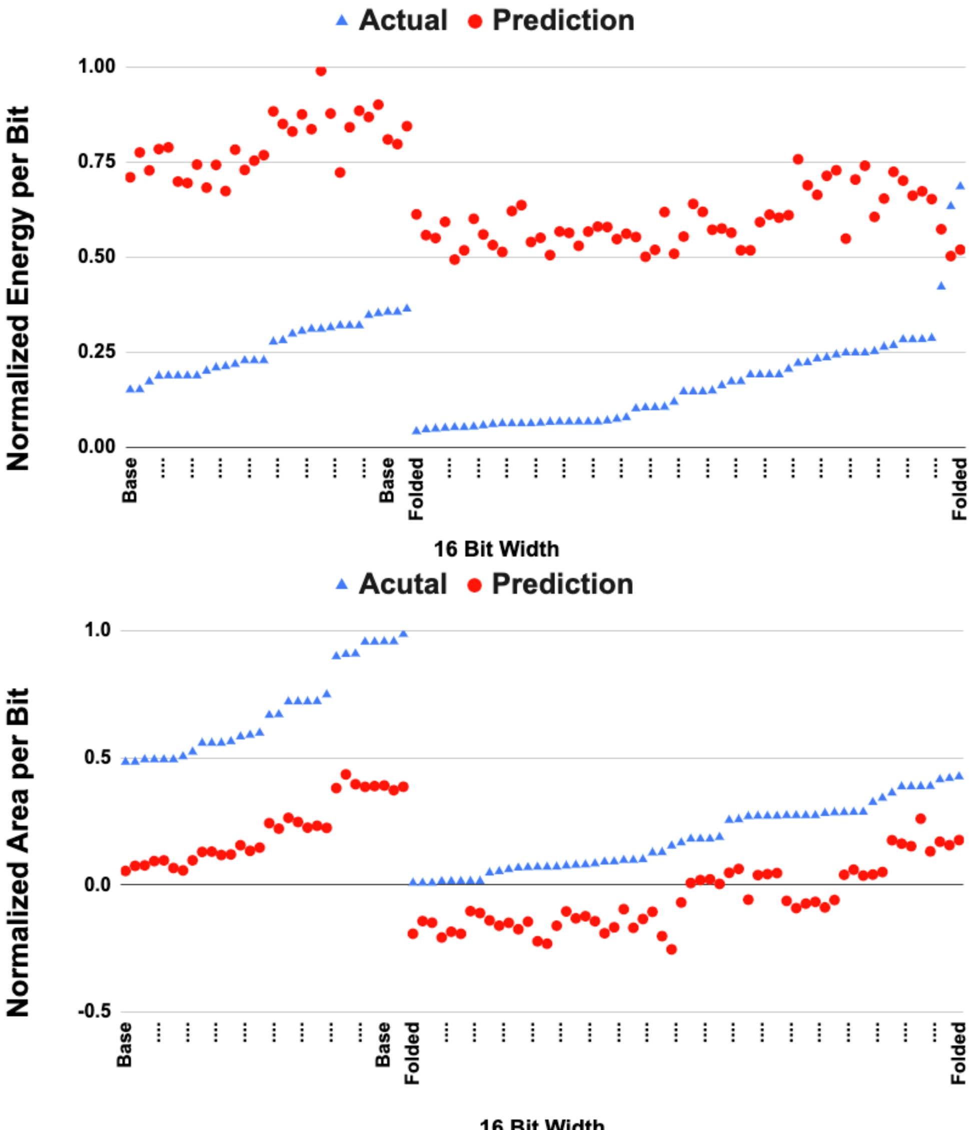


Figure 5.7: This plot shows predicted and actual values for normalized energy per bit (top) and normalized area per bit (bottom) for bespoke DCT accelerators. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend, indicating that our model can be used to predict the optimal architecture.

benchmark	Optimal architecture	Optimal Adder and Multiplier	Predicted Rank out of 174
binSearch	bm32	carry_select_adder_cell benc_radix8	11
div	openMSP430	carry_select_adder_cell nand_radix4	7
inSort	openMSP430	carry_select_adder_cell nand_radix4	7
intFilt	bm32	bounded_fanout_adder benc_radix8_mux	1
mult	bm32	bounded_fanout_adder benc_radix8_mux	5
tea8	bm32	bounded_fanout_adder benc_radix8_mux	5
tHold	openMSP430	bounded_fanout_adder benc_radix8_mux	7

Table 5.6: Summary of optimal architecture in terms of **energy per bit**; the model predicts with 100% accuracy in top 10% of predictions

5.5.2 Design Space Exploration For Bespoke DSP Accelerators

Figure 5.7 presents results for bespoke accelerators for the 32-bit Discrete Cosine Transform (DCT). The architectural parameter space spans folded and un-folded designs and all the architectural configurations discussed in Table 5.3. The figure plots normalized per-bit energy and area for different bespoke accelerators for an application that requires 16-bit input precision. Similar to the results for general purpose processors, although there is an offset between the predicted and actual metric values, the rank ordering of the predicted values follows the trend established by the actual values.

Table 5.7 presents the minimum-area bespoke accelerator generated using the bespoke methodology and its rank as predicted by our model. For each DSP accelerator, the number of variants generated is different, because the number of baseline architecture variants for each DSP accelerator circuit is different (see Table 5.2). To account for this difference, the predicted rank columns for each DSP accelerator are presented with reference to the total number of circuits evaluated. The predicted rank of the best design is within the top 10% of the predicted designs, with a few exceptions. However,

in the few cases where the optimal architecture is not in the top 10% of predictions, the actual metric values for the predicted and actual optimal architectures are still very close.

Just like for general purpose processors, the best architecture for a DSP accelerator can vary based on the application input precision and chosen efficiency metric. For example, the optimal FIR variant from which to generate a bespoke accelerator for an application with 4-bit input precision is a folded architecture, but for an application that requires 32-bit precision, the best variant is a pipelined architecture. Also, the optimal arithmetic unit architectures are different for these application scenarios; a bounded-fanout adder with a Booth-encoded radix-8 multiplier is best for a 4-bit input, and a conditional sum adder with a NAND-based radix-4 multiplier is best for a 32-bit input. Furthermore, although it is not shown explicitly in the results, the optimal architecture before bespoke customization is different than the optimal architecture after bespoke customization.

Similar to the results above, Table 5.8 presents the minimum-energy bespoke design and its predicted rank (shown with respect to the total number of architectural configurations). The results again confirm that with a few exceptions, the predicted rank of the optimal architecture is within the top 10% of candidates identified by our model.

5.5.3 Final Remarks

Across all the designs, general purpose processors and DSP accelerators, our models had an accuracy of ~88% in placing the optimal design in the top 10% of the search space. Our model was able to place architectural configurations in the top, middle, and bottom buckets with an accuracy of 91% for area and 88% for energy for general purpose processors and 88% for area and 87%, respectively, for DSP accelerators. Finally, using design space exploration, we were able to observe power and area saving improvements of up to 82% and 83%, respectively, 12% and 27%, respectively, on average, compared to a bespoke processor generated from the baseline design.

Table 5.7: This table presents the optimal architectural variant for each bespoke accelerator and its rank as predicted by our model for **area**. In most cases, our model ranks the optimal architecture within the top 10% of candidate architectures.

Design	4-bit input		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	cond_sum_adder & nand	15/58
DCT	folded	cond_sum_adder & nand	11/116
IIR	base	cond_sum_adder & radix4	2/58
L1	base	carry_select_adder_cell & nand_radix4	2/58
L2	base	cond_sum_adder & _radix4	2 /58
Sobel	base	carry_select_adder_cell & nand_radix4	2/58
FIR	folded	bounded_fanout_adder & benc_radix8	2/174

Design	8-bit input		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	cond_sum_adder & nand	8/58
DCT	folded	cond_sum_adder & nand	11/116
IIR	base	hybrid_adder & benc_radix8_mux	9/58
L1	base	carry_select_adder_cell & nand_radix4	2/58
L2	base	cond_sum_adder & radix4	2/58
Sobel	base	cond_sum_adder & nand_radix4	1/58
FIR	folded	bounded_fanout_adder & benc_radix8	2/174

Design	16-bit input		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	cond_sum_adder & and_radix4	2/58
DCT	folded	cond_sum_adder & nand	16/116
IIR	base	hybrid_adder & benc_radix8_mux	10/58
L1	base	carry_select_adder_cell & nand_radix4	2/58
L2	base	cond_sum_adder & and_radix4	2/58
Sobel	base	carry_select_adder_cell & nand_radix4	2/58
FIR	base	hybrid_adder & benc_radix8	1/174

Design	32-bit input		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	hybrid_adder & nand_radix4	8/58
DCT	folded	cond_sum_adder & nand_radix4	10/116
IIR	base	cond_sum_adder & and_radix4	1/58
L1	base	carry_select_adder_cell & nand_radix4	3/58
L2	base	cond_sum_adder & and_radix4	2/58
Sobel	base	cond_sum_adder & nand_radix4	1/58
FIR	pipeline	cond_sum_adder& nand_radix4	6/174

Table 5.8: This table presents the optimal architectural variant for each bespoke accelerator and its rank as predicted by our model for **energy**. In most cases, our model ranks the optimal architecture within the top 10% of candidate architectures.

Design	4-bit input		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	brent_kung_adder & and	5/58
DCT	folded	brent_kung_adder & nand_radix4	5/116
IIR	base	cond_sum_adder & and_radix4	2/58
L1	base	brent_kung_adder & nand	7/58
L2	base	hybrid_adder & nand	15/58
Sobel	base	brent_kung_adder & nand_radix4	45/58
FIR	base	hybrid_adder & benc_radix8	1/174
Design	**8-bit input**		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	brent_kung_adder & and	4/58
DCT	folded	brent_kung_adder & nand_radix4	2/116
IIR	base	cond_sum_adder & benc_radix8	1/58
L1	base	brent_kung_adder & and	9/58
L2	base	hybrid_adder & nand	13/58
Sobel	base	brent_kung_adder & nand_radix4	42/58
FIR	base	hybrid_adder & benc_radix8	1/174
Design	**16-bit input**		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	brent_kung_adder & and	22/58
DCT	folded	brent_kung_adder & nand_radix4	1/116
IIR	base	cond_sum_adder & benc_radix8	2/58
L1	base	brent_kung_adder & and	11/58
L2	base	hybrid_adder & nand	5/58
Sobel	base	hybrid_adder & and	1/58
FIR	base	hybrid_adder & benc_radix8	1/174
Design	**32-bit input**		
	Variant	Best Design's Adder & Multplier	Pred. Rank
Butterfly	base	ling_adder & benc_radix8	9/58
DCT	base	brent_kung_adder & and_radix4	5/116
IIR	base	carry_select_adder_cell & nand_radix4	5/58
L1	base	cond_sum_adder & and	9/58
L2	base	brent_kung_adder & and_radix4	7/58
Sobel	base	brent_kung_adder & and	5/58
FIR	pipelined	cond_sum_adder & nand_radix4	6/174

5.6 Generality And Limitations

The methodology presented in this paper quickly predicts the rank of an architectural configuration for a bespoke design with respect to other possible architectures for the same baseline design. To accomplish this, features were extracted from the baseline design and application binary, since the goal of our methodology is to predict the quality of an application-specific bespoke processor. However, the goal in a typical design flow is to optimize the design for power, area, and performance, irrespective of the application that will be run on the processor. Our methodology can be extended to a traditional flow by disregarding application-specific features while training our model.

Our methodology was evaluated on embedded processors and DSP accelerators. However, for larger more complex designs such as superscalar processors, multi-core processors, GPUs, or deep learning accelerators our methodology may need to be augmented with more complex and advanced models, such as a deeper model with more layers or graph convolutional layers to extract local/global connectivity information about the design's architecture and microarchitecture. Similarly, many more microarchitectural parameters can be extracted as features to train our model. For example, various structural widths (fetch, dispatch, execute, commit, etc.), forwarding path configuration, branch predictor size and design, cache configuration, etc. can be used to train a model to explore the design space. In such systems, a richer set of metrics can be targeted to train our model. For example, a designer may only be interested in improving the L1 cache hit rate or prefetcher accuracy instead of overall efficiency. Similarly, a designer may also be interested in not only predicting metrics such as power, energy, area, and performance but also metrics such as the maximum temperature attainable by a processor architecture while running a real application. By using a proper model, our methodology could conceivably be used to quickly explore the design space for these metrics and identify candidate architectural configurations that could optimize these metrics.

5.7 Related Work

5.7.1 Design Space Exploration

Design Space Exploration (DSE) for processor architectures has been significantly explored in prior art. Authors in [45] discuss microarchitecture optimization of the Intel Pentium Pro processor by tuning various microarchitectural parameters, such as pipeline length, cache size, and load store ports. [46] discusses a framework for exploring the design space of low-power application-specific programmable processors (ASPP), in particular media processors. The core idea of this work is reliance of high-quality compilers that exploit instruction-level parallelism (ILP) and reliable instruction-level simulators with modifiable architectural parameters such as issue width, size of cache, and number of execution units. Using their framework, they believe a designer could quickly evaluate the quality of an architecture for a set of applications that can be simulated on the simulator to evaluate power and area trade-offs of an architecture. Another work [47] presents techniques based on hill climbing, genetic algorithm, and ant colony optimization for design space exploration.

While the above works discuss design space exploration for single-core processors, several DSE techniques have been proposed for multi-core processors. Authors in [48] and [49] propose techniques to pose the multi-core architecture design space exploration problem as a multi-objective optimization problem and use evolutionary algorithms to explore and identify pareto-optimal solutions. Further [50] and [51] explore techniques for design space exploration in a single ISA heterogeneous chip multiprocessor setting. Going beyond multi-core processors, authors in [52] develop an optimization framework for a setting where multiple chiplets are used to build multiple systems, targeting different subset of applications.

While all the above techniques discuss design space exploration techniques for optimizing metrics such as power, area, and performance for processors and application domains, they do not explore the effect of producing the best design for a single application by trimming logic that is unusable by the application.

5.7.2 Application-Specific Processor Cores And High-Level Synthesis

One of the closest related work to our paper would be Extensible processors such as Xtensa [53], where a designer can specify configurations including structure sizing, optional modules (like debug and exceptions), and custom application-specific functional units. While this methodology enables a designer to generate a custom processor targeting QoR metrics such as power, performance, and area, this methodology does not allow for generating a custom processor for a single application at the granularity of logic gates. Several other techniques explore the space of application-specific processor core generation such as [54] and [55] that automatically develop hardware implementations connected to a general-purpose processor at the data cache and target compute-heavy parts in the workload. Such cores, while improving energy efficiency and power may not be area efficient, especially in ultra-low-power and area settings. Chip Multiprocessor Generators [56] allow a designer to generate different families of chips from scratch based on the application domain. However this requires domain expertise and knowledge which may not be automatable. These techniques still do not trim a processor at the finest granularity of gates.

High-Level Synthesis offered by tools such as Cadence Stratus [57] and Siemens Catapult [58] do produce a custom ASIC for a given C program. However, the process of HLS can be significantly slower and more expensive, since the high-level specification of the application behavior still needs to be specified, and this specification itself needs to be verified. In contrast, optimizing an already-verified core for a single application in an automated fashion can significantly reduce design costs.

5.8 Summary

In this chapter, we have presented a novel methodology that quickly explores the design space of architectural configurations of a hardware design and predicts the configuration that will produce the optimal application-specific bespoke design for a particular target application and metric. Our methodology uses machine learning to train a neural network on various features extracted from the application binary, base hardware design, and architectural configurations to predict a metric of interest. For a given target application, we use the predicted metric for each architectural configuration to identify

near-optimal candidates for more detailed evaluation and ranking.

Our evaluations show that for all GPP designs evaluated, the true optimal architectural configuration is in the top 10% of the predicted ranks. For DSP accelerators, except for a few cases, the top 10% predicted architectural configuration contained the optimal architecture. In the few exceptions, the top 10% contained at least one near-optimal architectural candidate. Overall, our model had an accuracy of ∼88% in identifying the optimal design within the top 10% of the search space. Our model was able to place architectural configurations in top, middle, bottom buckets with an accuracy of 91% for area and 88% for energy, for general purpose processors. For DSP accelerators, our model was able to place the designs in the right buckets with an accuracy of 88% for area and 87% for energy. Finally, we showed that by exploring the architectural design space we can improve power and area savings by up to 82% and 83% for power and area, over generating a bespoke design from the baseline design. On average, we showed that the power and area savings of exploring the architectural design space over all designs and applications were 12% and 27%, respectively.

In this work, we explored the vastness of architectural parameter space and identified near-optimal architecture for a given application optimized for a given metric. We also successfully navigated through the non-linear relationship between the impact of bespoke methodology and the processor-application features. Handling these problems with human intuition and creativity alone would have been challenging. There is also the problem of enormous simulation time and manual effort. Machine learning proved a useful tool in effectively handling these challenges. This also motivates the application of machine learning to other complex computer architecture problems. In the next chapter, we will discuss how we extended the principles of symbolic simulation, design space exploration, and hardware-software co-design to develop a bespoke domain-specific processor for Secure Multi-Party Computation systems.

Chapter 6

Bespoke Domain-specific Architecture for Secure Multi-Party Computation

In today's data-driven world, privacy and ownership of data are of paramount importance. There is significant economic value associated with personal data [59]. The global value of private data was estimated to exceed $3 trillion in 2019, and is growing at an unprecedented rate, expected to exceed $7 trillion by 2025 [60, 61]. Large corporations are hesitant to share the data that they own but are invested in making business decisions based on insights from combined datasets belonging to two or more corporations. Also, extremely-protected data, such as medical records in healthcare that are protected by HIPAA regulation, would need to pass significant regulatory scrutiny before sharing with third parties for analysis and research. Secure Multi-Party Computation (MPC) systems enable new computational paradigms in which multiple parties jointly compute a function on their private inputs without revealing any information about their inputs to each other.

Software solutions for MPC [1, 62] involve evaluating logic as encrypted logic gates, which involves significant overhead, and communication bottlenecks. Existing hardware acceleration technologies for MPC include garbled circuit-based implementations, which have significant network and hardware overhead. The current state-of-the-art includes

software-based circuit evaluation, which uses XOR-based secret shares to overcome the network challenges of garbled circuit-based implementations. However, significant overheads associated with software-based solutions make them infeasible in most practical applications, preventing widespread use of MPC in applications where it could otherwise have a revolutionary impact.

To address these challenges and enable MPC for more applications, we sought to apply bespoke optimization techniques to customize a general-purpose processor (GPP) for MPC applications. However, because the computational paradigm for MPC is radically different than the GPP paradigm, a bespoke customization of a GPP fails to overcome the bottlenecks inherent in MPC applications sufficiently to make MPC a feasible solution for most applications. With this understanding, we designed a novel ISA and microarchitecture for MPC, resulting in a new *template* from which bespoke MPC designs can be generated to achieve orders of magnitude improvements in performance and energy efficiency for MPC applications.

However, the benefits made possible by our domain-specific bespoke MPC architecture can only have a significant impact if the software development flow for our processor is accessible to normal programmers. To this end, we have developed a software toolchain for MPC processors that provides a simple python programming interface, compiler, and assembler that generate performance-optimized executables for python applications.

6.1 Introduction to Multi-partly Computation

Secure multiparty computation (MPC) is a cryptographic protocol that allows multiple parties to jointly compute a function on their private inputs without revealing any information about their inputs to each other. This is useful for a variety of modern data-intensive applications, including the following use cases.

Data analytics: Secure multi-party computation (MPC) enables the analysis of distributed data without granting any single party access to data owned by other parties. This technique finds utility in various domains, including fraud detection, risk assessment, and market research. Consider the scenario where fintech companies aim to enhance their understanding of the market's loan repayment capability. By leveraging

MPC, these companies can construct models that incorporate account information for individuals across multiple banks, enabling more accurate and comprehensive market analysis.

Machine learning: MPC can be used to train machine learning models on private data that are distributed across multiple parties. This can help to improve the accuracy, generality, and reduce prediction bias of the models, as they are trained on larger and more diverse datasets

For instance, imagine a collaborative effort among healthcare institutions to develop a predictive model for identifying early signs of a particular disease. Each institution holds a portion of the relevant patient data, including medical records, genetic information, and diagnostic test results. Through the application of MPC, these institutions can securely collaborate and jointly train a machine learning model without directly sharing sensitive patient information. By combining their datasets through MPC, the resulting model benefits from a larger pool of diverse data, leading to improved accuracy and better prediction capabilities. Furthermore, since MPC maintains privacy of the sensitive datasets, regulatory burden to protect and securely access the datasets can be essentially eliminated, enabling much more extensive research and development of healthcare related models involving Protected Health Information(PHI).

Privacy-preserving computation: MPC can be used to compute functions on data while inherently protecting the privacy of the data owners. This can be useful for tasks such as medical diagnosis, financial transactions, advertisement targeting, and voting. In the context of advertising, consider a platform such as Facebook that possesses valuable data on individual social profiles. Meanwhile, a product company desires to promote ads on this platform. In order to optimize their advertising campaign and achieve the highest return on investment, the product company requires additional user information that the platform is unwilling to disclose due to privacy concerns. Similarly, the product company possesses data on user interactions with their own product and conversion details, which could potentially enhance the ad platform's ability to target ads more effectively. In this scenario, both parties can employ secure multi-party computation (MPC) techniques to develop models that generate only the final prediction without directly accessing sensitive user information, enabling both parties to build better targeting models to improve their return on investing.

MPC is a powerful tool that can be used to protect data privacy while still allowing the data to be used productively. As the amount and value of data that are generated and stored continues to grow, MPC will become increasingly important for ensuring data privacy and security.

Currently, software-based solutions are the most commonly-used implementation of MPC. Software-based MPC solutions have the advantage of being easily deployable on a general-purpose computing platform. However, they suffer from extreme performance bottlenecks due to the complexity of the underlying cryptographic protocols and the large cost associated with exchanging a substantial amount of data between the parties [63–66]. This is particularly true when dealing with large-scale computations involving many parties, which can result in prohibitive computational and communication overheads.

To overcome these limitations, there is a growing interest in exploring the potential of hardware-based solutions for MPC [63, 66]. Hardware-based solutions can offer significant performance advantages compared to software-based solutions by leveraging the capabilities of specialized hardware, such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs). These hardware devices can be optimized for specific MPC protocols, allowing for higher computational and communication efficiency.

One of the key advantages of hardware-based MPC solutions is their ability to perform computations in parallel, which can significantly reduce the overall computation time. In addition, hardware-based solutions can minimize communication overhead by integrating communication components directly into the hardware, reducing the latency and power consumption associated with data transfer. Furthermore, hardware-based solutions that are designed with security in mind, providing additional protection against side-channel attacks and other security threats could be eliminated because MPC is immune to such security vulnerabilities by design.

6.2 Background on MPC

XOR secret sharing is a fundamental building block of many MPC systems, which enables multiple parties to perform collaborative computations on their private data

without revealing that data to each other. In these systems, the parties each hold a share of a secret, and they collaborate to compute some function on their private data by sharing their shares, then performing a computation using the shares. In XOR secret sharing, each party holds a share that consists of a random value. To reconstruct the secret, the parties collectively perform an XOR operation on their shares.

XOR secret sharing can be used in a wide range of multiparty computation systems, including secure function evaluation, secure multi-party computation, and secure two-party computation. The security of these systems relies on the fact that no subset of parties smaller than a threshold can learn anything about the secret, and that the parties can perform computations on the secret without revealing anything about their private data to each other.

6.2.1 Logic implementation using XOR-Secret Share(XOR-SS)

In MPC systems, parties can cooperate to compute several functions. To allow a full variety of functions, the system should be functionally complete. A functionally complete set of logical connectives is one that can be used to express any possible function by combining members of the set into a Boolean expression. There are many different sets of operations that are functionally complete. For example, one common functionally-complete set of logic connectives is {AND, OR, NOT}; any logic function can be realized using only these three connectives. XOR-SS uses the functionally-complete set {XOR, AND} to implement an arbitrary logic. The reason for this choice among the many options is that the MPC XOR function can be computed independently and privately without the need for communication among the parties during computation (Section 6.2.1). Computing the AND function, however, requires communication between the parties while the computation is performed (Section 6.2.1).

XOR: Communication Free

The XOR function can be computed in MPC without the need for data exchange between the parties during computation. The basic idea is that each party contributes their share to the computation, and the shares are combined in such a way that the secret can be reconstructed only if a sufficient number of parties participate. For example, suppose that two parties – Alice and Bob – each hold a share of a secret and wish

to compute the sum of their private values. Figure 6.1 illustrates how this computation can be done, where Alice owns 0xA4, Bob owns 0x2B, and the goal is to compute 0xA4 XOR 0x2B without revealing the values to the other party.[1] The steps are as follows.

1. Each party splits the value they own into XOR-secret shares by XORing their value with a random number (R_a and R_b, respectively).

2. Each party exchanges a part of their share with another party.

3. Each party independently performs the XOR function on the data.

4. The final result is opened, or revealed, by XORing the resulting shares.

It is important to note that the logic computation itself could be performed independently by each of the parties without any communication during the computation, because the XOR operation is reversible [67]. The effect of the random numbers R_a and R_b goes away when the final results of the parties are XORed together while revealing the result (open to party, as shown in Figure 6.1); thus, XOR computation can proceed locally without any external communication during the computation to another party. An illustrative example is shown below.

$$share_a^1 = A \oplus R_a \tag{6.1}$$

$$share_a^2 = R_a \tag{6.2}$$

If we perform an XOR operation again with the R_a we can recover back the A.

$$A = share_a^2 \oplus share_a^1 \tag{6.3}$$

However, a similar logic does not hold for AND operations as performing an AND operation with a random bit, would lose information about the input value. Even if we know one of the inputs and the output we cannot recover the value. Therefore, AND is not a reversible operation. Since the AND operation is not reversible, it requires

[1] This is only an illustrative example to demonstrate how the XOR function is performed. Since each party only owns one value, it would be possible for either of the parties to learn the value of the other party by performing the XOR operation with the value they own and the final revealed result.

back-and-forth communication to be interleaved with the computation, as discussed below.

AND: Communication Required

As noted in the previous section, XOR can be computed without sharing any information because the XOR operation is reversible [67]. However, this is not true for the AND function, and thus, parties must communicate intermediate terms to complete an AND function. One way to perform this computation is by using Beaver triples [68].

A *Beaver triple* is a tuple of three values (a, b, c), where a and b are random shares of two inputs, and c is the XOR of the product of the two inputs and a random value r. Beaver triples can be precomputed and shared among the parties, so that they can be used as building blocks for evaluating Boolean circuits. Figure 6.1 describes the procedure for generating secret shares using Beaver tiples [1].

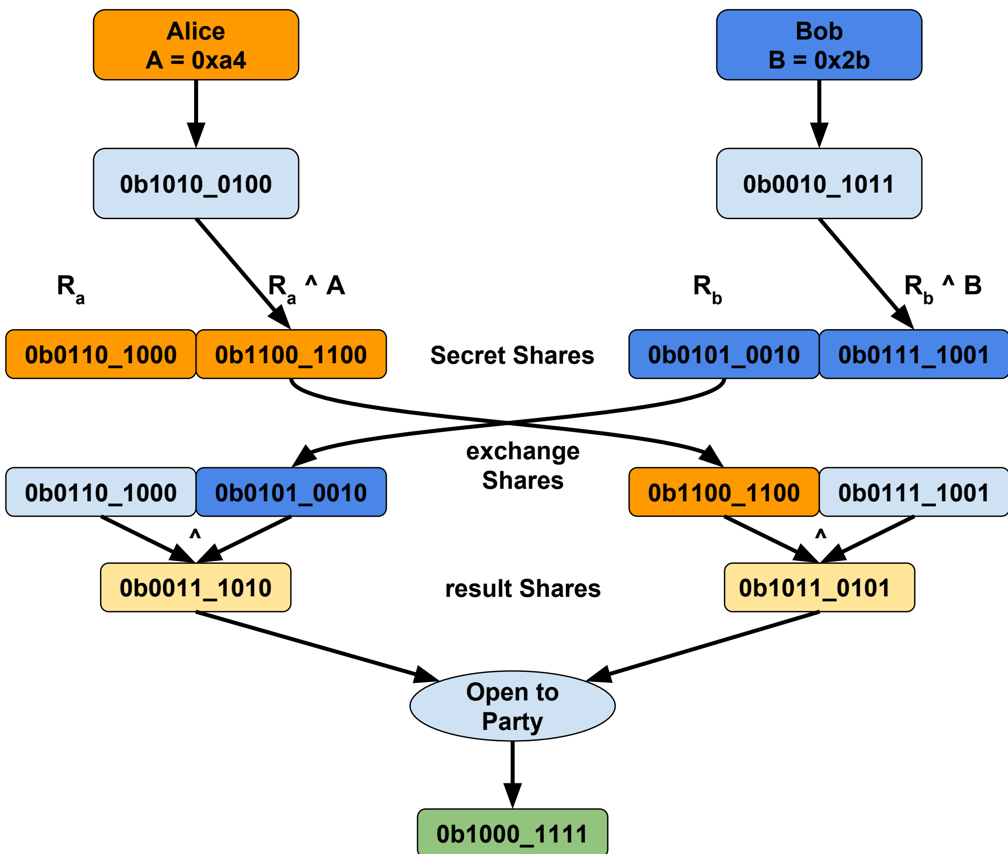


Figure 6.1: The XOR function can be computed without interleaving communication with the computation. In this example, Alice and Bob own 0xA4 and 0x2B, respectively. XOR is computed by generating secret shares, exchanging the shares, and having each party compute XOR independently. Finally, the result is revealed by opening the shares.

Algorithm 4 Implementing AND logic using Beaver triples

1: Alice generates random value u and shares it with Bob using XOR-SS.
$\{u_1$ is with Alice and u_2 is with Bob$\}$
2: Bob generates random value v and shares it with Alice using XOR-SS.
$\{v_1$ is with Alice and v_2 is with Bob$\}$
3: Alice selects a triple (a_1, b_1, c_1).
4: Bob selects a triple (a_2, b_2, c_2).
$\{a, b, c \mid a = a_1 \oplus a_2,\ b = b_1 \oplus b_2,\ c = a \cdot b\}$
5: Alice computes: $u_1 \oplus a_1$, shares it with Bob.
Alice computes: $v_1 \oplus b_1$, shares it with Bob.
6: Bob computes: $u_2 \oplus a_2$, shares it with Alice.
Bob computes: $v_2 \oplus b_2$, shares it with Alice.
7: Alice receives $u_2 \oplus a_2$ and $v_2 \oplus b_2$, and computes
$w_1 \leftarrow c_1 \oplus (((u_1 \oplus a_1) \oplus \underline{(u_2 \oplus a_2)}) \cdot b_1) \oplus (((v_1 \oplus b_1) \oplus \underline{(v_2 \oplus b_2)}) \cdot a_1) \oplus (((u_1 \oplus a_1) \oplus \underline{(u_2 \oplus a_2)}) \cdot ((v_1 \oplus b_1) \oplus \underline{(v_2 \oplus b_2)})))$
{underlined terms are obtained from the other party}
8: Bob receives $u_1 \oplus a_1$ and $v_1 \oplus b_1$, and computes
$w_2 \leftarrow c_2 \oplus (((u_2 \oplus a_2) \oplus \underline{(u_1 \oplus a_1)}) \cdot b_2) \oplus (((v_2 \oplus b_2) \oplus \underline{(v_1 \oplus b_1)}) \cdot a_2)$

9: **Output:** The output of the AND gate is the XOR of the values computed by Alice and Bob, i.e., $w_1 \oplus w_2 \equiv u \cdot v$

As seen in Algorithm 4, 2-bit information is exchanged between the two parties to realize the AND operation. In general, $2N$ bits would be exchanged in an N-party system. Because it requires considerably more communication than XOR, the AND operation is considered relatively expensive.

6.3 Related Work

Existing MPC approaches are implemented as custom logic circuits, constructed from a functionally-complete set of security-enhanced logic gates. The underlying cryptographic techniques involve translating an application into an equivalent circuit representation that ensures the privacy and security of a collaborative computation among multiple parties while preserving the confidentiality of each party's inputs. At this time, there exist two fundamental approaches for MPC – Garbled Circuits (GC) [69] and Secret-Sharing (SS) [70]; both involve representing and evaluating an application as a Boolean circuit constructed from a functionally-complete set of security-enhanced gates {XOR, AND}. The circuit-based approach supports the composition of complex functions by combining smaller circuits or gates. This enables the construction of more intricate computations using a modular and scalable approach. Homomorphic Encryption presents another way of performing private computation; however, the approach is limited to one party, and as such is out of scope for this work.

Garbled Circuit (GC) in the cloud using enabled nodes [71] provides an end-to-end implementation of GC on the cloud that includes a garbler and an evaluator implemented on separate FPGA nodes. Garbled circuits, in general, are computationally expensive, as the truth table of the gates is encrypted. E.g., SHA-1 is used to encrypt the trruth table in [71]. Additionally, the accelerator is provided with a gate evaluation table, and the hardware evaluates a fixed circuit. This approach does not scale well, expecially for problems in which an entire application cannot be laid out as a stream of logic gates at compile time.

Hardware-Software Co-Design to Accelerate Garbled Circuits (HAAC) [66] presents a co-design strategy aimed at enhancing GC performance. This is achieved through the utilization of a customized compiler and dedicated hardware accelerator. The HAAC compiler facilitates the expression of GCs as multiple streams, enabling parallel processing of GCs through specialized hardware units known as gate engines (GEs). The implementation of HAAC relies on the Bristol netlist [72] generated by the EMPToolkit – a toolkit known for its comprehensive capabilities in secure computation [62].[2]

[2]Bristol format and Bristol fashion are standard file formats used for representing digital logic circuits.

The evaluation of gates in the HAAC approach follows a breadth-first approach, wherein a dataflow path is assessed on dedicated hardware. However, as the application size grows, the number of gates also increases, resulting in significant overhead associated with wireId and encrypted gateId management. The inherent inefficiency of building custom logic circuits for an application and evaluating them involves significant bookkeeping overheads like loading the circuit graph, evaluating the gate, and propagating them. This bookkeeping overhead becomes more pronounced, potentially impacting the evaluation of each gate within the system.

Private Computation Framework (PCF) is an XOR Secret Share-based implementation of MPC [1] overcomes the fixed circuit limitations of prior work [71] by taking a just-in-time style approach for compilation. In PCF, an MPC application is compiled into a circuit using the MPC implementations of XOR and AND gates (a functionally-complete set), and a software-based technique is used to evaluate the circuit. Applications are commonly implemented in C++ and leverage private data structures provided by the PCF. Evaluation of applications in PCF takes place through an MPC backend, which functions akin to a virtual machine. PCF provides an open-source MPC implementation that is used in the industry to perform MPC computation over advertisement data. The approach involves several significant overheads. For one, a just-in-time (JIT) compilation approach is used to translate an application into an equivalent circuit, which adds significant latency in the computation. For circuit evaluation, every individual secure bit is indexed by a 64-bit wireId; therefore, representing a 64-bit value requires 64 64-bit wires. The transmission of intermediate tuples also incurs this additional overhead. The circuit is formed by overloading C++ operations$(+, -, |, \&, >, <, >=, <=)$, and the application itself is written as a heavily-templated C++ application. The application code must be compiled together with the entire library framework before execution, and the initial phase of execution involves the translation of the application into a circuit, as described above. The circuits are first formed at runtime, then topologically sorted, and finally, gates are evaluated through software emulation. Each of these steps involves significant overhead. Another potentially-limiting requirement is that the entire circuit must fit in the main memory.

6.4 Bespoke Processors for Secure MPC

Due to the challenges inherent in performing collaborative computation while maintaining the privacy of all parties involved, current state-of-art MPC implementations are subject to various limitations, including significant communication bottlenecks, scheduling inefficiency for circuit evaluation, runtime overhead due to dependency on expensive JIT-like compilation, memory and execution overhead management, inefficiency of existing circuit evaluation techniques, and domain expertise needed for custom MPC application specification. While many existing applications could benefit (sometimes quite significantly) from secure MPC, the limitations of existing approaches preclude its use for most applications. In this work, we propose a novel MPC architecture that addresses the limitations of existing MPC approaches, with the goal of enabling MPC for a much wider range of applications. In this section, we describe the proposed MPC architecture, along with the software toolchain that we developed to allow average developers to write MPC applications without extensive domain-specific expertise. Below, we provide a comprehensive exploration of the design and functionality of the compiler, assembler, instruction set architecture (ISA), and hardware microarchitecture constituting the overarching framework.

6.4.1 Compiler and Assembler design

One substantial bottleneck that currently prevents widespread adoption of MPC is the learning curve required to develop an MPC application. Due to their inherent complexity, designing MPC applications using any of the current state-of-art approaches demands a considerable time investment. Furthermore, the current infrastructure necessitates building the entire framework alongside the specific application being developed. This approach is less than ideal, as it introduces significant dependencies for distribution [1]. In light of the shortcomings of prior implementations, there is a need for governing rules that ensure ease of usage, easy adaptation, and scalability. To this end, we follow the strategies below in the design of our secure MPC framework.

First, we provide a Python-based application frontend for developers to write applications, allowing easy adoption. Although Python is an interpreted language, we only use its syntax and fundamental data types for compilation. We add a few syntax rules,

outlined in Listing 6.1, to assist the compiler in generating efficient MPC code. These rules, which are used to specify the bitwidth of a data value, help the compiler and assembler identify required hardware units to optimize application performance. One reason for these rules is that when operating with secret shares, there is no way to assess the bitwidth of the data type, as the values are XOR-encrypted. This approach simplifies the application writing process for developers who do not want to learn about the framework for describing front-end applications. We leverage python's language parser and syntax tree generator [73, 74] to implement our compiler.

Listing 6.1: Rules for frontend declarations

```
# For Assign statements declare datatype like so:
# @mpc: DType::<TYPE>
a = 42

# Example for Bit/Boolean
# @mpc: DType::kBit
a = 1

# Example for 8-bit
# @mpc: DType::kByte
b = 0x23

# Example for 16-bit unsigned
# @mpc: DType::k2Byte
c = 0x8004

# For public data types, `i` is set to public and `d` is inferred
    from `i`
# @mpc: DType::kPublic
d = i

# For expressions with consts, const takes the Type of
   destination
# Limit to 1 destination operand
```

```
a[i] = c[i] + 95

# Supported data types
DType {kBit, kByte, k2Byte, k4Byte, k8Byte, kPublic}
```

Second, we provide the ability to evaluate vectored expressions. In existing MPC implementations [1, 66], every individual bit is represented by a separate Bit type variable, regardless of whether or not the bit is batched with others. This approach leads to suboptimal processor or emulator utilization, since tracking each bit requires significant overhead. To address this issue, our compiler aims to reduce this overhead at both the emulator and hardware levels by using symbols that consist of a value and its status as the base type (see Section 6.4.2 for details), thereby minimizing the bookkeeping overhead. Since prior work accesses each bit separately, a processor with 64-bit addressing and 64-bit registers would need 64×64 bits to track one 64-bit value. Since we track by symbols and their internal bit-status, our implementation incurs only 64 bits of status overhead for every value.

Third, we reassemble the execution graph with the aim of achieving parallelism. Prior work on PCF [1] simply evaluates expressions at the gate level, making the concept of read/write dependency immaterial. With such an implementation, we can identify the independent logical paths by performing a simple topological sort and issuing independent objects in parallel. However, with our higher-level description of the application in Python language, we must ensure that control and data dependencies are handled correctly. Algorithm 5 describes in more detail how the compiler handles and manages the dependency graph.

Finally, we generate an Intermediate Representation (IR) of the datapath so that the assembler can perform hardware-specific optimizations based on the hardware configurations and constraints. Using an IR allows us to optimize machine code for different bespoke MPC hardware implementations. Some of the hardware optimizations we enable are described in Section 6.4.3. Once we have resolved all control and dataflow dependencies, we can segment every datapath into threads up until the point where they converge. This approach simplifies the process of achieving parallelism and significantly enhances scalability.

We define the high-level compilation flow for our proposed approach in two steps.

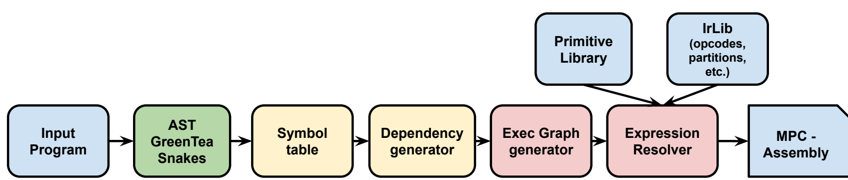


Figure 6.2: High-level flow for compiling a front-end application in Python into an MPC-friendly representation enabling the assembler to generate optimal machine code.

The first step involves parsing and generating an intermediate representation of the input program. The next step is to generate machine code for the intermediate representation by mapping and assembling the MPC ISA instructions. The overall compilation flow is depicted in Figure 6.2. The assembler further optimizes the machine code generated in the previous step to encourage communication reduction by efficiently using the hardware structures and caching intermediate results, with the goal of improving the overall performance of the system. Algorithm 5 describes the process of generating the assembler-friendly intermediate representation.

6.4.2 MPC ISA

The need for a new ISA in the context of MPC arises due to certain limitations of current ISAs. First, certain instructions, such as arithmetic operations, are inherently bound to be computed bit by bit, necessitating communication between the parties for each bit to make progress. For instance, for an N-bit operation, an N-round trip would be required for an ADD instruction that uses a ripple-carry adder backend, resulting in a significant communication overhead. This holds true for any instruction that uses non-free gates[3].

Second, current ISAs have limitations in handling such instructions, since the overall number of registers is limited, and it is inefficient to keep swapping the working set with the cache or lower level memory to compute N-bit operations. Note that in MPC, we may not obtain the result for an instruction in one cycle, due to the requirement to perform communication for every bit. Therefore, a new ISA is needed to address these

[3] *Non-free gates* refers to a gate or a group of gates that contains one or more AND gates and thus requires communication to evaluate.

Algorithm 5 High-Level Flow for Compiling Front-End Application into MPC-friendly Intermediate Representation

1: Initialize symbol table
2: Generate initial syntax tree using Python AST
3: Extract abstract syntax tree
4: Break down the graph into sections
5: **for all** sections **do**
6: Build symbol table of variables
7: Generate dependencies for section based on discovered entry point
8: identify independent statements
9: Build execution graph for section using dependency graph and reordering
10: Generate groups of instruction that could be evaluated in parallel
11: Assign group ID and instruction ID
12: Perform interpretation of expressions
13: Assign primitive types necessary to evaluate the expression and set statuses of the symbols based on the dependency graph and symbol scope
14: Reorder the instructions
15: **end for**
16: Reorder *sections*
17: **Output:** Intermediate representation of the instructions that can be consumed by the assembler to generate the MPC machine code and assembly listing

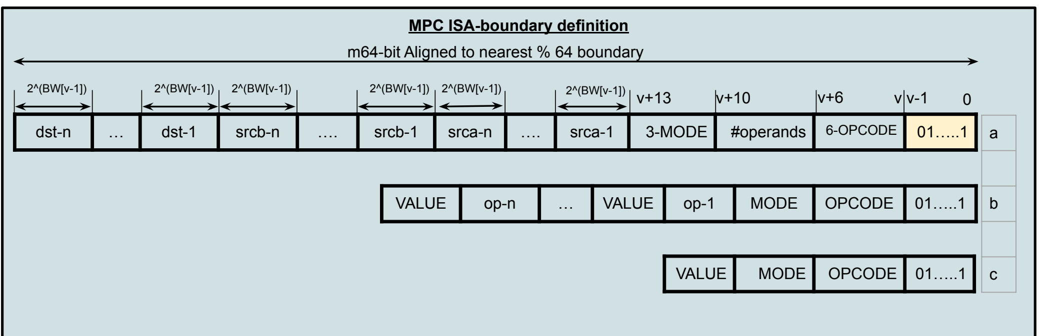


Figure 6.3: Encoding and boundary description for different classes of instructions that allows vectored instruction with efficient packing.

limitations and enable efficient computation in MPC.

A larger working set comprising registers and caches that can track the status of each bit is required. This approach would enable scheduling of portions of instructions as data become available. We propose a symbols table that is analogous to the symbol table generated by a compiler. This symbol table would be partitioned for each section of the program space, comprising loops, functions, and programs, based on the granularity of branching within the high-level application code.

Our work introduces a new encoding scheme that facilitates variable-length instruction coding. Figure 6.3 presents an outline of different instruction classes and their boundaries. The encoding scheme is designed to support multiple operands simultaneously and enable the specification of operand bitwidths using MODE bits. The encoding scheme is outlined below.

- The yellow block in Figure 6.3 represents a variable-length thermometer-encoded variable that specifies the length of the instruction in multiples of 64-bits.
- The OPCODE is a 6-bit field[4], and its encoding varies based on the opcode in use. Currently, 32 instructions are supported, with headroom provisioned to add an additional 32.
- The #operands field occupies four bits, and the MODE field consists of three

[4]A 6-bit opcode is chosen so that the instruction set can be extended.

bits that specify the bitwidth of each operand. The srcA, srcB, dstA, and dstB fields all have the same bitwidth – 2^{MODE}. The MODE field plays an important role in the computation optimization, as it allows certain arithmetic operations to be limited to a smaller bitwidth, thereby reducing the amount of multi-party communication required to perform an operation.

- `MODE=0b111` is a special case in which the operands are considered public, and thus, execution of the instructions does not require any communication; execution follows a normal execution pattern. Therefore, the `MODE` specifies the granularity to work on datasets anywhere from 1-bit to 64-bit, providing the compiler with flexibility to aggressively perform optimization to minimize communication and enhance performance where necessary.

- We incorporate a VALUE field that is reserved for certain opcodes, such as fencing or synchronization instructions.

- All the operands specified in the class A.x and B are symbol-ids that are addressed by the preset addressing mode of the compiler. Therefore, based on the complexity of the underlying hardware and application, this option can be configured at the compiler level. Currently, we have experimented with a symbol table that is 16-bit addressed. As such, bitwidth is four.

Our goal is to provide an efficient and flexible instruction encoding scheme that can enhance the overall performance of the instruction set. Instructions are further divided into classes, as described below.

- **Class A.1:** **3-operand**: Logical and arithmetic operations

- **Class A.2:** **4-operand**: A special case of A.1 for ternary operations, arithmetic operations with N, Z, C, V update

- **Class B:** **2-operand**: Mainly for memory and control flow operations such as {Load, Store, Set, Reset, jumps, and conditional branches}

- **Class C:** **no-operand**: Instructions such as fence and nop, used for synchronization.

We have identified several key insights related to instruction processing in the context of MPC, which we list below.

1. To improve performance, instructions that can potentially run in parallel are grouped together. The instruction fetch logic is configured to create instruction groups which are then dispatched to parallel units for execution.

2. However, there can be dependencies between different instruction groups, such as Read after Write (RAW) or Write after Read (WAR). To address these, we identify a limited number of symbols on which such dependencies exist. In cases where there is a circular dependency, groups may not proceed, and barriers must be added to govern when operations can proceed. In such situations, we use a *Fence* instruction between the two groups.

3. Conditional branching is a special case of execution that can be realized via instruction predication. In the case of encrypted results, for instance, the result of `a > b` in Figure 6.4 would be encrypted, and there would be no way to determine which execution path should be chosen, making it impossible to determine which group of instructions to execute in the conditional block. We address this issue by using instruction predication to select the value determined by the result of the condition, which would still be encrypted. A detailed implementation of this approach is discussed in Section 6.4.3 on our hardware architecture.

4. Finally, certain instructions require all bits to be processed before a control decision can be made, such as conditional statements. E.g., to evaluate `(a+b) > 5`, we must compute all non-revealed results of the sum, then compute the results of the final (a+b-5) to evaluate the condition of the result. To address such situations, we use `Fence` instructions to ensure that all the bits of one instruction group are computed before the next group can be scheduled. There could be many statements within an *if* block. As shown in Figure 6.4, the result of the condition `a > b` is common for the two statements and can be determined by the N, Z, C, V flags after performing $a - b$. Thus, the processor needs to wait for $a - b$ to complete before the results can be assigned to `r, t` as N, Z, C, V flags will not be ready until $a - b$ is complete.

`if (a > b){`
`  r = s;`
`  t = b;`
`} else {`
`  r = q;`
`  t = a;`
`}`
	`r = a > b ? s : q;`
	`t = a > b ? b : a;`

Figure 6.4: Conditional branching statements are grouped as ternary operations based on the condition; the assembler implements $>, <, >=, <=$ as a part of a subtract operation where the N, Z, C, V flags can be reused to evaluate different conditional outcomes without having to re-evaluate the conditional outcome for every different condition involving the same result. This reduces the amount of communication required.

By addressing these key points, we aim to provide an efficient and flexible instruction encoding scheme that can enhance the overall performance of the instruction set in the context of MPC. With this new approach, we provide a General Purpose Processor for MPC applications, allowing the same hardware be utilized for many MPC applications, as opposed to building, optimizing, and evaluating a custom circuit for each application, as in [1, 66, 71].

6.4.3 Hardware Architecture

The hardware architecture for MPC must enforce synchronization between parties involved in the computation for every bit in the non-free gates. To manage the communication mechanism between the parties, we introduce an abstraction called *Conduit*. Similar to a regular processor architecture, there are Fetch, Decode, Execute, and Write Back stages in the pipeline abstraction. However, the fetch unit attempts to group instructions until a fence is encountered, assuming that the compiler manages the ordering of instructions such that they can be executed in parallel without circular dependencies until the fence instruction. It is important to note that hardware resources should not be exhausted, i.e., an appropriate stalling mechanism is needed to observe structural dependencies. If there are m adders and m addition instructions without circular dependencies, all m instructions can be scheduled in parallel. However, for subsequent scheduling of add instructions, the fetch unit must stall or attempt to re-order instructions. The decode and fetch stages work cooperatively until a circular dependency is detected and all instructions are fetched into a group. The compiler and assembler are

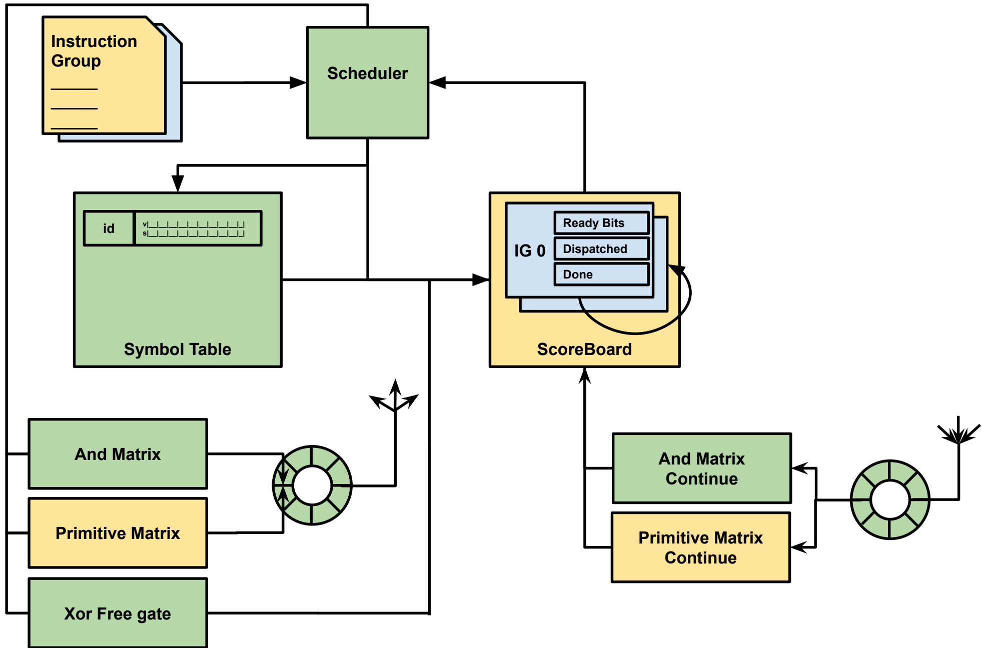


Figure 6.5: Architecture diagram of the execution unit. The scheduled instruction that needs communication is put on the conduit in order and transmitted to the other party. Upon receiving the partial terms from the other party, corresponding operations continue their operation. Finally, upon completion, the status bits are updated in the scoreboard.

responsible for ordering the instructions such that groups are formed easily; although the intermediate representation of the instruction contains group-id, they are not packed into the machine code, and hardware forms its own groups. Once a group is dispatched, the progress of the group in terms of computation is managed at the group level rather than the instruction level, which reduces bookkeeping overhead.

Scheduler

In the Execute unit, much of the specialized hardware design and customization for MPC occurs. Since the system gives the ability to group the instructions we would not need multiple units to track the status of the groups. Therefore, as shown in Figure 6.5, the execution unit is designed with a scoreboard that tracks the progress of each bit

of a hardware instruction group. The unit is broken down into sub-modules, with the scheduler responsible for routing the operands to the necessary processing element, and each processing element is responsible for putting its partial results on the Conduit so that the total buffer order is maintained. As the partial computations are received from the other party, the operation continues, and finally, the done bits are updated in the scoreboard. Algorithm 6 depicts the overall scheduling algorithm and how the scheduler plays its role in the life of an instruction and/or an instruction group. Note that done_queue is cleared once the section of the instruction goes out of scope or the same instruction is rescheduled.

The Conduit Mechanism

All operations that involve AND as a part of their computation require communication. As discussed earlier, there is a dedicated Processing Element matrix that we call the Primitive matrix; this block involves gate-level implementation of several structures to support both arithmetic and logical operations.

Every structure that involves non-free gates would need to communicate intermediate terms to the other party by serializing the data. Based on the structure type and the structure ID, partial results are inserted into a circular buffer to preserve the total order of operations. As shown in Figure 6.5, the receiver conduit can receive its share of the partial result asynchronously. It is important to note that the receive order is known and guaranteed. Upon receiving the partial result from the other party, the operation will continue for each primitive in the order it was queued to the conduit.

The transmitter side of computation can be decoupled from the `continueOperation` circuit; this would incur additional storage overhead and require a context switcher. Decoupling would allow scheduling more operations while other operations are in transit and/or some other operations are continuing their operation.

Hardware Optimization

Below, we briefly discuss some of the main primitives from which other primitives are derived and designed.

Adder and Subtractor: ADD and SUB instructions, apart from serving the purpose of addition and subtraction, also serve as the base for many conditional instructions.

Algorithm 6 MPC scheduler algorithm

```
1: waiting_queue ← initialize empty Queue()
2: running_queue ← initialize empty Queue()
3: done_queue ← initialize empty Queue()

4: // Decoder puts the instruction group in the waiting queue
5: for all inst_group in waiting_queue do
6:   load/create scoreboard entry
7:   inst_group→reconfigureReadyBits()
8:   // configures the status bits for the instruction group to identify if any instruction
     could be scheduled.
9:   if bitCount(inst_group→readyBits) > 0 then
10:    for all inst in inst_group do
11:      if inst is not done then
12:        status ← assign processing element
13:        //If the operation is not complete and needs communication
14:        if status in ScheduleStatus::kNeedsComm then
15:          //Processing Elements route the intermediate results to the Conduit,
             scheduler does not handle this logic.
16:          running_queue→add(inst)
17:        else if status in SchedulerStatus::kSuccess then
18:          //Operation finished for the scheduled bits successfully, update status
19:          updateDstXStatus()
20:        end if
21:      end if
22:      updateGroupStatus()
23:    end for
24:    group_status ← getGroupStatus()
25:    //check if all bits of all the instructions are completed
26:    if group_status in GroupStatus::kDone then
27:      done_queue→add(inst_group)
28:    else
29:      waiting_queue→add(inst_group)
30:    end if
31:  end if
32: end for

33: //continue running the instructions if we received communication from other parties

34: for all inst_group in running_queue do
35:   inst_group→continueOp()
36: end for
```

Every instruction involves building a custom logic structure that ultimately resolves into a combination of AND and XOR gates. However, careful design considerations should be in place to select appropriate structures. For example, an N-bit ripple-carry-based implementation would require N serial carry computations. This may be suboptimal when compared with a parallel structure like the kogge-stone adder [75] or brent-kung adder [76] which would take (Log-N) steps to converge. However, one should note that with an increase in bitwidth, the number of cascaded AND operations for the generate and propagate phases of these adders also increases significantly. Based on Table 6.1, which compares the AND gate complexity of various adders, we can come to the conclusion that although the RCA has a longer chain, it has the least amount of AND operations, and therefore reduces overall communication bandwidth requirements. Thus, we choose RCA for our implementation. One may wonder how an N-bit RCA can be realized using only N-AND gates. In general, we have the following formulation.

$$sum_i = a_i \oplus b_i \oplus c_{i-1} \tag{6.4}$$

$$c_i = a_i.b_i + a_i.c_{i-1} + b_i.c_{i-1} \tag{6.5}$$

This requires three AND gates and two OR gates. However, carry can be re-written as:

$$c_i = ((a_i \oplus c_{i-1})(b_i \oplus c_{i-1})) \oplus c_{i-1} \tag{6.6}$$

This reduces the overall AND usage to one gate per bit. Also, as the hardware supports pipelining an arithmetic instruction; if an output bit from the previous instruction is used as input in the next instruction, it is automatically forwarded. This is particularly possible if the output can be obtained bit by bit. However, for divide operation, this is not possible because the progression of getting quotient and reminder is not sequential from least significant bit to most significant bit.

Another optimization that is baked into the assembler is the tracking of N, Z, C, and V flags. These flags are particularly necessary not only to track the outcome of arithmetic instructions like `ADD/SUB`, but also to determine the outcome of a conditional branch. However, tracking these flags itself requires additional logic. For instance, the

Z flag is computed as {~|`sum`}, which translates to (N-1) OR[5] gates and a negation of the result, this adds to the overall overhead of the circuit. The assembler can identify that the flags will not be necessary based on application constraints, and the instruction could be mapped to a hardware unit that does not compute these flags.

MUX: We implement conditional instructions using `MUX` as a primitive, similar to [1]. The reason for choosing such a strategy is that the result of the condition will not be known even at run-time, as the values would be XOR-encrypted. As a result, the condition would not be known, making it impossible to execute a branch instruction in a traditional sense. However, to handle such a situation, we use the ternary operator which, as shown in Figure 6.4, essentially translates to a 2:1 MUX. The MPC implementation is described as follows:

$$result = (compositeAND((A \oplus B), choice) \oplus A) \tag{6.7}$$

In Equation (6.7), choice is a single bit, while A and B are N-bit words. As shown in Algorithm 4, we would need four bits of data to be exchanged between the parties if we are computing one AND result. However, since the choice bit is common, we can generate special tuples with fixed b, such as $(a_1, a_2).b = (c_1, c_2)$, and thus, we would only need $n + 1$ bits to be transferred from each party, we call this AND operation a *compositeAND*. However, the generation of tuples and oblivious transfers are out of the scope of this research.

4-bit Adder	#AND Gate	#OR operation	Total ANDs
Ripple Carry	4	0	4
Brent Kung	20	12	32
Kogge Stone	14	6	20

Table 6.1: Analysis of AND gate complexity with different Adders

Oblivious RAM (ORAM) [77] is one of the ways to securely perform memory accesses without knowing the physical address/location of the storage in memory. Although our ISA supports memory instructions, in the current implementation we omit ORAM. [78] provides one of the better approaches to implement ORAM.

[5] OR gate is implemented as $(a.b) \oplus (a \oplus b)$

6.4.4 MPC processor description

In this section, we provide a description of our bespoke MPC processor, which we evaluate using a cycle-accurate in-house simulator written in C++17. The key characteristics of our processor are as follows.

- The processor architecture employs a 5-stage pipeline, consisting of Fetch, Decode, Issue, Execute, and Write-back stages.
  - The Fetch stage incorporates an elastic fetch unit capable of extracting multiple instructions until a fence instruction is encountered.
  - The Decode stage works in conjunction with the fetch stage to generate instruction groups.
  - The Issue stage is responsible for creating symbol table entries and dispatching instructions to the execute stage.
  - The Execute stage features a scoreboard that manages the overall execution status of a group, facilitating the dispatch of instructions to the corresponding processing elements, namely the XOR-matrix, AND-matrix, and primitive library.
- Our evaluation focuses on the 8-bit mode of the processor, employing a 16-bit addressing mode for symbols, and utilizing a symbol table with a size of 64kB.

6.5 Methodology

We have implemented a custom cycle-accurate hardware emulator for our bespoke MPC processor.[6] Our emulator is written in C++, and it can execute the MPC machine code generated by our compiler, which is written in pure Python3.10. We evaluate the performance of our MPC processor and compare our results against the current state-of-art MPC implementation – PCF [1]. PCF also uses XOR secret share-based MPC, so we are able to make an apples-to-apples comparison of application performance between the two MPC execution frameworks.

[6] We will release our emulator and software toolchain to the public to encourage further research on MPC and private computation.

6.5.1 Baseline setup

We setup the PCF framework, use benchmarks that are already available in the codebase, and develop some additional applications, mentioned in Section 6.5.2. Our setup consists of the machine configuration mentioned in Table 6.2. The programs and the framework were compiled in a Docker container [79], and results for a two-party setup were obtained on the same clustered machine, so as to avoid network communication latency.

Property	Value
Architecture	x86_64
Processor	Intel(R) Xeon(R) CPU E5-2670 v3
Cores per socket	12
Sockets	2
Threads per core	2
Frequency	2.30GHz (max-3.1GHz)
RAM	128 GB

Table 6.2: Baseline configuration

6.5.2 BENCHMARKS

We investigate various benchmark programs commonly utilized in Multi-Party Computation (MPC) applications. One of the benchmarks we consider is the standard billionaire program, which is a variant of Yao's millionaire program [80]. This particular benchmark has been widely recognized and employed as a standard reference within the MPC community, as noted in the codebase of [1]. Additionally, we have implemented several data processing application benchmarks; namely Sum, Count, and ReLU [81]. These benchmarks serve as representative examples for evaluating the performance and capabilities of MPC systems. For further details and descriptions of the benchmark programs, please refer to Section 6.5.2.

6.6 Results

In this section, we present the performance evaluation of our custom MPC hardware for the benchmarks described in Section 6.5.2. We present our analysis in terms of

Table 6.3: Benchmark Programs

Program	**Description**
Billionaire	Alice and Bob wish to determine who has the greater net worth (cash + stocks + property) without revealing their exact net worths to each other. The problem consists of 2 vectors containing 32 elements of type `struct {uint8_t cash, uint8_t property, uint8_t stock}` named `alice` and `bob`. The goal is to compute a boolean array where `(alice[i].cash + alice[i].property + alice[i].stock) > (bob[i].cash + bob[i].property + bob[i].stock)`.
Sum	In this benchmark, we reduce a vector of 32 elements by summing them. This program explores looping and the write-after-read technique for the `sum` variable.
Count	This program counts how many times a certain value in the array is greater than a specified value. Again, the array contains 32 elements. It involves branching statements that essentially test if the hardware can evaluate conditional statements correctly.
ReLU	ReLU, or the Rectified Linear Unit, selects `a[i] = max(0, a[i])` for 32 elements.

run time compared with the industry standard MPC framework – PCF [1], network usage analysis, and its implication to performance. Finally, we discuss overall resource utilization and its correlation to runtime.

6.6.1 Performance variation with network speed

MPC applications in general depend immensely on a stable network. Hence, large corporations tend to co-locate the data on the third-party cloud where they can leverage slightly higher intra-cloud network speeds. As the data is already encrypted through XOR-SS, parties can leverage this network advantage by using a common third-party cloud. Figure 6.6 shows for the Billionaire benchmark [1, 80] how the network speed influences the overall computation. In our design, we choose 400MHz frequency for the following reason: we have one communication channel, and based on the variation plot, there would not be significant performance improvement beyond reaching the maximum communication bandwidth, *network wall*.

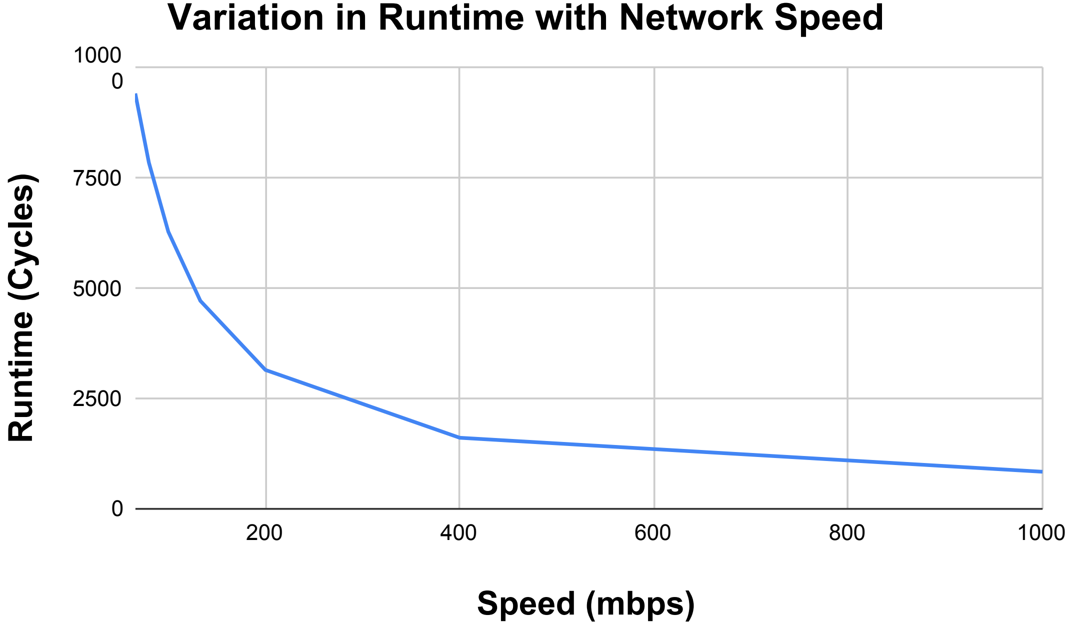


Figure 6.6: The performance of MPC computation is directly correlated to network speed. As the network speed gets closer to the frequency of our hardware, network no longer becomes the bottleneck. This plot is for the processor running at 400MHz.

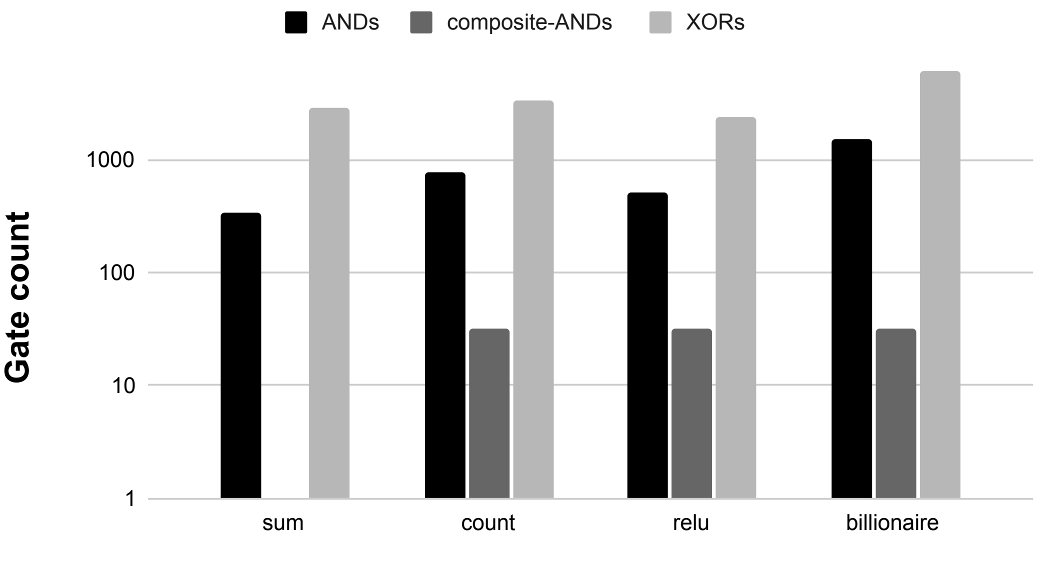


Figure 6.7: Resource utilization: shows the total number of AND, XOR, and Composite AND operations that were scheduled for different benchmarks.

Table 6.4: Benchmark results shows the runtime of various benchmarks to compute and final reveal. Reveal here refers to decrypting the final result by sharing the result shares and performing the XOR operation on the result.

Benchmark	Time (ns)	After reveal(ns)	Tx traffic	Tuples	ANDs	Composite ANDs	XORs
sum	5187.5	5283.5	344	344	344	0	2838
count	12027.5	12123.5	800	1024	768	32	3296
relu	8657.5	11729.5	544	768	512	32	2368
billionaire	23557.5	25093.5	1568	1792	1536	32	6080

6.6.2 Comparison with baseline

In prior work [1], there are certain optimizations like topological sorting of the logic gates before evaluation so that independent gates could be evaluated in parallel depth-wise, allowing batching of variables. However, it is important to acknowledge that regardless of whether we are evaluating a bit vector or an integer, PCF treats each individual bit as a separate operand. Consequently, this approach leads to the evaluation of each bit independently, which represents a significant suboptimal utilization of modern processors. It is worth mentioning that a 64-bit processor evaluates only one bit at a time, not to mention the additional overhead caused by bookkeeping data structures associated with the Bit type.

Nonetheless, there are valid reasons why PCF has been adopted. It simplifies gate evaluation and facilitates the management of graph topo-sorting. Furthermore, it facilitates communication across different components in a more manageable manner. However, the significant overheads incurred by PCF translate to substantial CPU and Memory inefficiency. Also, most of the frameworks available for MPC require the whole of the circuit, be it XOR-based or Garbled Circuit-based [64, 69], require that the entire circuit layout be present in the main memory. This represents a particularly prohibitive overhead for large data problems. First, the high-level logic should be synthesized and laid out on software gates, and then evaluated. This process adds orders of magnitude in overhead. The overheads come from first, building the initial circuit which involves translating logic into gates and assigning gate and wire Ids. Next, sorting and reordering the circuit. Finally, the execution of the circuit at the software level which also involves communication and synchronization with the other parties.

Our implementation significantly outperforms the state-of-art PCF for all benchmarks, as seen in Figure 6.8. In general our approach outperforms the software implmentation [1] by $\sim$**20000**$\times$ in runtime as with our approach we decouple the compilation of the circuit with our MPC compiler. Alongside reaping the benefits of the compiler, we get additional performance benefits by significantly reducing the software overheads that stem from having to manage a laid-out boolean circuit. We can also compare our performance with [66], where the authors claim around 800x improvement over Emptoolkit implementation on a similar setup as our baseline. Our implementation improves performance by an additional 10$\times$.

Figure 6.9 shows the network utilization and the total number of tuples consumed, benchmarks *count, relu* and *billionaire* incurs relatively lesser network traffic compared to the number of tuples consumed as these benchmarks have one or more conditional statements allowing the use of *compositeAND* operation, *compositeAND*s reduces the number bits to be transferred as discussed earlier. Hence, improves the overall network utilization. Network usage, in general, depends on the number of `ADD, SUB`, and conditional instructions, since the *sum* benchmark reduces an array of 32 elements by summing, it has no conditional branching and a relatively less number of `ADD` instructions compared to other benchmarks. Therefore, it generates the least number of network traffic among the benchmarks.

Table 6.4 shows the resource usage for each of the applications. Runtime depends on the number of AND gates, as seen from benchmarks *sum* and *relu*, the relative difference in runtime is higher for *count* as opposed to *sum*. This is because the comparison operation(which includes `sub, lt, mux`) should be complete before the increment can be updated in the *count* benchmark. Note, that in the increment operation, the destination of the `add` instruction is one of the source operands. Therefore there is a circular dependency, where we would have to wait to schedule this instruction until the result is updated. CompositeAND gates are specifically useful in evaluating conditional instructions, as they involve the MUX operation to pick a result. Thus, we can notice that *sum* does not use any composite AND gates, as it does not have any conditional statements.

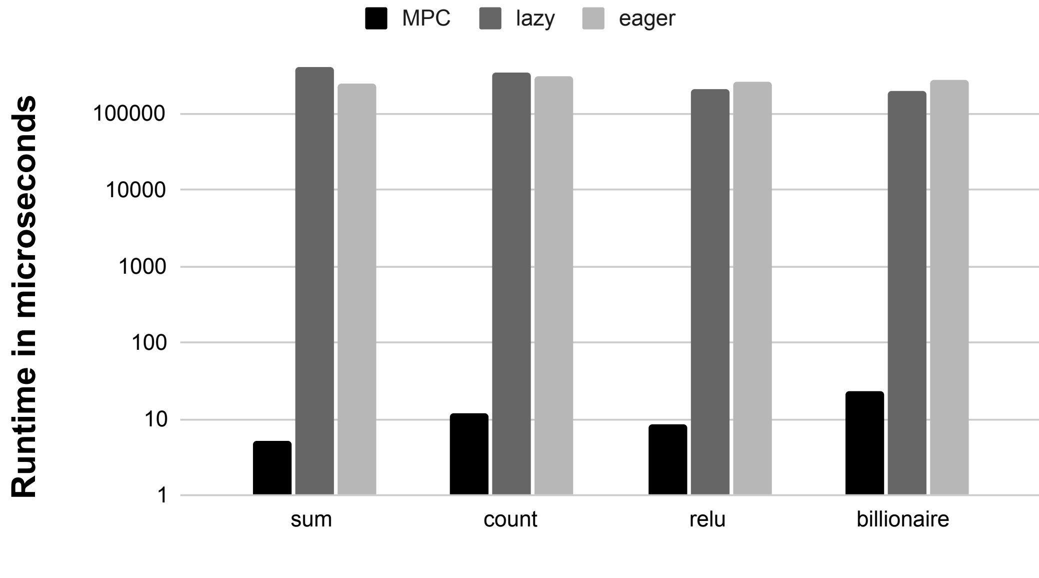


Figure 6.8: Shows the variation of runtime in microseconds for various benchmarks with the baseline results from the PCF platform [1] consisting of two schedule modes – Eager and Lazy.

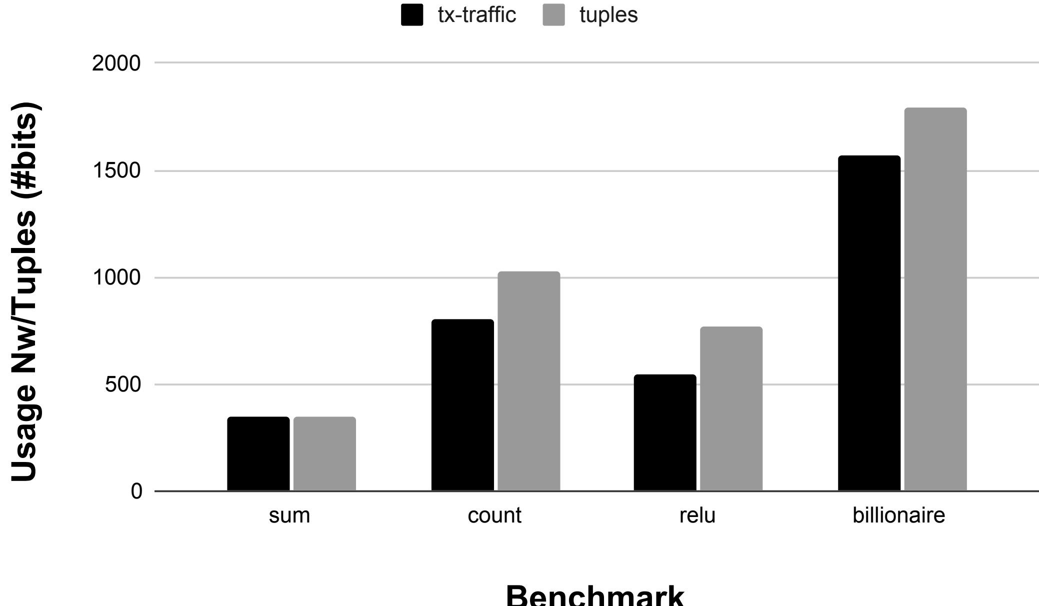


Figure 6.9: Shows the network usage and the number of tuples used for various benchmarks. Tuples used and network traffic mainly depend on the number of *AND* and *compositeAND* operations. Except for *sum*, the rest of the benchmarks consists of conditional instructions which use the *compositeAND* gate; thus, the tx-traffic is reduced.

6.7 Conclusion

Secure Multi-Party Computation (MPC) systems enable new computational paradigms in which multiple parties jointly compute a function on their private inputs without revealing any information about their inputs to each other. In this chapter, we present a complete hardware architecture and software toolchain for MPC, including compiler, assembler, and customized processor architecture that facilitates the seamless implementation of Secure Multi-Party Computation. The novel solution proposed in this chapter exhibits superior performance compared to existing alternatives, offering a comprehensive approach from the foundational level to the end result. To the best of our knowledge, this represents the first-ever hardware/software solution presented in the field. Numerous avenues for further improvement exist, encompassing both performance enhancement and security augmentation. A promising direction for scaling the hardware entails promoting vectored processing through dedicated hardware design, leveraging the existing compiler support. Additionally, incorporating hardware mechanisms for secure memory access via Oblivious RAM stands out as a logical progression for future exploration.

Chapter 7

Conclusion And Discussion

Emerging applications like wearables, implantables, and IoT applications are characterized by ultra-low area and power requirements, and they run the same software over and over, as defined by their application. Recent works have shown that symbolic simulation-based hardware-software co-analysis enables application-specific hardware optimizations that can result in significant area, power, and energy savings. The co-analysis technique uses symbolic simulation to mark gates that are exercisable by the application for some application input. However, the technique treats the application as a black box, and hence, suffers from the pessimism of marking too many gates as exercisable, potentially leaving significant benefits on the table. In this work, we showed that incorporating program semantics in the form of application constraints into the co-analysis technique defines application behavior more accurately and better optimizes the hardware for area, power, and energy efficiency. We described the means to statically analyze an application binary, form constraints for commonly occurring code patterns, and enforce the constraints in the gate-level simulation.

Further, we built a design-agnostic simulation tool that enables application-specific hardware optimizations on any design, technology, or architecture. Prior works built a custom simulator that tailors one specific processor design for a given application. To allow application analysis on an arbitrary design, we modified iverilog – a verilog synthesis and simulation tool – to perform symbolic simulation-based hardware-software co-analysis. We demonstrated the generality of our tool by performing symbolic co-analysis for three microprocessors with different ISAs.

With the generality of our hardware-software co-analysis tool, we opened up the scope to modify the architecture of a processor and allow application-specific analysis of the new design, whereas prior works considered processor architecture to be fixed. Considering the enormous architecture parameter space and the significant synthesis and simulation time required to analyze all possible designs, we built an ML-based tool that takes into account the impacts of application-specific optimizations on different architectural features and predicts a near-optimal architecture for an application with respect to a metric of choice. This tool allows us to limit the detailed synthesis and simulation of designs to a select few near-optimal options and thus expedites the architectural exploration process.

Application-specific knowledge lead us to design a bespoke domain-specific processor architecture for MPC applications. The domain-specific architecture is designed in a general-purpose fashion, with a standard application programming framework, making it compatible with our symbolic approach to capture application information and use that application information to automatically optimize bespoke processors for specific MPC applications. Furthermore, our MPC software toolchain can generate executables for custom bespoke processors for specific applications. Our domain-specific bespoke architecture for Secure Multi-Party Computation addresses some of the critical challenges in the domain, such as computation and communication overhead, as well as programmability and accessibility. Our bespoke architecture provides a simplified Python programming interface and abstracts away the need for an MPC application developer to have in-depth knowledge of the inner workings of MPC. Our solution includes a compiler, assembler, and bespoke processor architecture that is specifically tailored to MPC applications. A promising direction for scaling the hardware entails promoting vectored processing through dedicated hardware design, leveraging the existing compiler support. Additionally, incorporating hardware mechanisms for secure memory access via Oblivious RAM stands out as a logical progression for future exploration.

With the generic symbolic simulation-based hardware-software co-analysis tool that we built, the main target architecture is ultra-low-power embedded systems that predominantly employ in-order processors. With further research, we can extend the co-analysis technique to out-of-order processors. For this, we must devise innovative techniques to handle symbol propagation to branch targets, prefetch addresses, etc. Other

difficult scenarios include handling dependencies through symbols, speculative execution, and more. By extending the co-analysis technique to out-of-order cores, we can analyze an application's accurate impact on the processor without having to rely on traces that are input-based, which is the current norm. Since most of the new innovations in computer systems are evaluated upon trace-based simulations before being implemented in hardware, replacing the input-based traces with symbolic simulation-based traces that more accurately and comprehensively represent application behavior can yield better and more impactful innovations.

To conclude, this dissertation extends symbolic simulation-based hardware-software co-analysis by introducing a more accurate generic tool that performs application-specific analysis and optimization on any design, technology, or architecture. This dissertation also advances architecture optimizations by exploiting the capabilities of machine learning techniques and applying bespoke design optimization to an important emerging application domain – secure MPC.

References

[1] Aditya Shastri, David Wu, Lynn Chua, Ruiyu Zhu, Tal Davidi, Yu Sheng, and Josh Mintz. Private Computation Framework 2.0. 2022.

[2] Paul Gerrish, Erik Herrmann, Larry Tyler, and Kevin Walsh. Challenges and constraints in designing implantable medical ics. *IEEE Transactions on Device and Materials Reliability*, 5(3):435–444, 2005.

[3] Seetharam Narasimhan, Hillel J Chiel, and Swarup Bhunia. Ultra-low-power and robust digital-signal-processing hardware for implantable neural interface microsystems. *IEEE trans. on biomedical circuits and systems*, 5(2):169–178, 2011.

[4] Michele Magno, Luca Benini, Christian Spagnol, and E Popovici. Wearable low power dry surface wireless sensor node for healthcare monitoring application. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 189–195. IEEE, 2013.

[5] Chulsung Park, Pai H Chou, Ying Bai, Robert Matthews, and Andrew Hibbs. An ultra-wearable, wireless, low power ECG monitoring system. In *Biomedical Circuits and Systems Conference, 2006. BioCAS 2006. IEEE*, pages 241–244. IEEE, 2006.

[6] Kris Myny, Steve Smout, Maarten Rockelé, Ajay Bhoolokam, Tung Huei Ke, Soeren Steudel, Brian Cobb, Aashini Gulati, Francisco Gonzalez Rodriguez, Koji Obata, et al. A thin-film microprocessor with inkjet print-programmable memory. *Scientific reports*, 4:7398, 2014.

[7] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: system support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170, 2012.

[8] Ross Yu and Thomas Watteyne. Reliable, Low Power Wireless Sensor Networks for the Internet of Things: Making Wireless Sensors as Accessible as Web Servers. *Linear Technology*, 2013.

[9] G. Hackmann, Weijun Guo, Guirong Yan, Zhuoxiong Sun, Chenyang Lu, and S. Dyke. Cyber-Physical Codesign of Distributed Structural Health Monitoring with Wireless Sensor Networks. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):63–72, Jan 2014.

[10] K. Myny, E. van Veenendaal, G. H. Gelinck, J. Genoe, W. Dehaene, and P. Heremans. An 8b organic microprocessor on plastic foil. In *2011 IEEE International Solid-State Circuits Conference*, pages 322–324, Feb 2011.

[11] BK Charlotte Kjellander, Wiljan TT Smaal, Kris Myny, Jan Genoe, Wim Dehaene, Paul Heremans, and Gerwin H Gelinck. Optimized circuit design for flexible 8-bit rfid transponders with active layer of ink-jet printed small molecule semiconductors. *Organic Electronics*, 14(3):768–774, 2013.

[12] Hari Cherupalli, Rakesh Kumar, and John Sartori. Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems. In *Computer Architecture (ISCA), 2016 43th Annual International Symposium on*. IEEE, 2016.

[13] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Enabling effective module-oblivious power gating for embedded processors. In *High Performance Computer Architecture, 2017. HPCA 2017. IEEE 21st International Symposium on*. IEEE, 2017.

[14] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori. Bespoke processors for applications with ultra-low area and power constraints. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 41–54, 2017.

[15] Randal E Bryant. Symbolic Simulation – Techniques and Applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 517–521. ACM, 1991.

[16] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Determining application-specific peak power and energy requirements for ultra-low-power processors. *ACM Trans. Comput. Syst.*, 35(3), December 2017.

[17] Shashank Hegde, Subhash Sethumurugan, Hari Cherupalli, Henry Duwe, and John Sartori. Constrained conservative state symbolic co-analysis for ultra-low-power embedded systems. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 318–324. IEEE, 2021.

[18] Mingoo Seok, Dongsuk Jeon, Chaitali Chakrabarti, David Blaauw, and Dennis Sylvester. Pipeline strategy for improving optimal energy efficiency in ultra-low voltage design. In *Proceedings of the 48th Design Automation Conference*, pages 990–995, 2011.

[19] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Software-based gate-level information flow security for iot systems. In *Procs. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 328–340, 2017.

[20] Stephen Williams and Michael Baxter. Icarus verilog: Open-source verilog more than a year later. *Linux J.*, 2002(99):3, jul 2002.

[21] Subhash Sethumurugan, Shashank Hegde, Hari Cherupalli, and John Sartori. A scalable symbolic simulation tool for low power embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 175–180, 2022.

[22] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146, 2009.

[23] A. Ibing and A. Mai. A fixed-point algorithm for automated static detection of infinite loops. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 44–51, 2015.

[24] W. Zuo, P. Li, D. Chen, L. Pouchet, Shunan Zhong, and J. Cong. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[25] Olivier Gerard. openmsp430, a synthesizable 16bit microcontroller core written in verilog. 2018.

[26] Synopsys. *Design Compiler User Guide.*

[27] Cadence. *Encounter User Guide.*

[28] Bo Zhai, Sanjay Pant, Leyla Nazhandali, Scott Hanson, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Todd Austin, Dennis Sylvester, et al. Energy-efficient subthreshold processor design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):1127–1137, 2009.

[29] EEMBC. EEMBC Benchmarks. `http://www.eembc.org`, 2020.

[30] Graph 500. `http://www.graph500.org`.

[31] Synopsys. *VCS/VCSi User Guide.*

[32] Sunil R Das, Sujoy Mukherjee, Emil M Petriu, Mansour H Assaf, Mehmet Sahinoglu, and Wen-Ben Jone. An improved fault simulation approach based on verilog with application to iscas benchmark circuits. In *2006 IEEE Instrumentation and Measurement Technology Conference Proceedings*, pages 1902–1907. IEEE, 2006.

[33] Wikipedia. List of wireless sensor nodes. `https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes`, note = "[Online; accessed 7-April-2016]", 2016.

[34] Jacob Borgeson. Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform. *Texas Instruments White Paper*, 2012.

[35] C. Roth, L.K. John, and B.K. Lee. *Digital Systems Design Using Verilog*. Cengage Learning, 2015.

[36] darklife. Darkriscv open source riscv implementation. 2021.

[37] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[38] Veni Mohan, Akhilesh Iyer, and John Sartori. Enhancing workload-dependent voltage scaling for energy-efficient ultra-low-power embedded systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[39] Nathan Bleier, John Sartori, and Rakesh Kumar. Property-driven automatic generation of reduced-isa hardware. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 349–354. IEEE, 2021.

[40] David J Wheeler and Roger M Needham. Tea, a tiny encryption algorithm. In *International workshop on fast software encryption*, pages 363–366. Springer, 1994.

[41] Synopsys. *DesignWare-Adder-Multiplier-Characterization.*

[42] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014.

[43] Amrut Kapare, Hari Cherupalli, and John Sartori. Automated error prediction for approximate sequential circuits. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2016.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] D.B. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[46] J. Kin, Chunho Lee, W.H. Mangione-Smith, and M. Potkonjak. Power efficient mediaprocessors: design space exploration. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 321–326, 1999.

[47] Sukhun Kang and Rakesh Kumar. Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization. In *2008 Design, Automation and Test in Europe*, pages 1432–1437, 2008.

[48] Horia Calborean and Lucian Vinţan. An automatic design space exploration framework for multicore architecture optimizations. In *9th RoEduNet IEEE International Conference*, pages 202–207, 2010.

[49] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, Chantal Ykman-Couvreur, Maryse Wouters, Carlos Kavka, Luka Onesti, Alessandro Turco, Umberto Bondi, Giovanni Mariani, Hector Posadas, Eugenio Villar, Chris Wu, Fan Dongrui, Zhang Hao, and Tang Shibin. Multicube: Multi-objective design space exploration of multi-core architectures. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 488–493, 2010.

[50] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, page 23–32, New York, NY, USA, 2006. Association for Computing Machinery.

[51] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 11–22, 2011.

[52] Saptadeep Pal, Daniel Petrisko, Rakesh Kumar, and Puneet Gupta. Design space exploration for chiplet-assembly-based processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):1062–1073, 2020.

[53] Cadence. Tensilica Processor IP. https://www.cadence.com/en_US/home/tools/ip/tensilica-ip.html.

[54] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1):205–218, mar 2010.

[55] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 163–174, 2011.

[56] Alex Solomatnikov, Amin Firoozshahian, Wajahat Qadeer, Ofer Shacham, Kyle Kelley, Zain Asgar, Megan Wachs, Rehan Hameed, and Mark Horowitz. Chip multi-processor generator. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 262–263, New York, NY, USA, 2007. Association for Computing Machinery.

[57] Cadence. *Stratus High-Level Synthesis User Guide.*

[58] Siemens. Catapult High-Level Synthesis. https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/.

[59] Alessandro Acquisti and Catherine Taylor. The economics of privacy. *Journal of Economic Literature*, 54(2):442–492, 2013.

[60] Vasudha Thirani and Arvind Gupta. The value of data. *World Economic Forum*, 2017.

[61] World Economic Forum. The future of jobs and skills: Employment, skills and workforce strategy to 2025. Technical report, 2018.

[62] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[63] T. Araki, T. Yamada, and Y. Yamaoka. Performance evaluation of secure computation technologies for the practical use of mpc-based private cloud. *IEEE Transactions on Dependable and Secure Computing*, 15:1096–1108, 2018.

[64] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, 1982.

[65] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.

[66] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. Haac: A hardware-software co-design to accelerate garbled circuits. *arXiv preprint arXiv:2211.13324*, 2022.

[67] J. W. Morgan. *Boolean Algebra and Digital Logic*, chapter 2, pages 58–59. Prentice Hall, Upper Saddle River, NJ, 5th edition, 2012.

[68] Michael O. Rabin and Yehuda Lindell. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 239–247, Burlington, VT, USA, 1996.

[69] Andrew C. Yao. How to generate and exchange secrets. *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[70] Amos Beimel. Secret-sharing schemes: A survey. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, pages 11–46, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[71] Kai Huang, Mehmet Gungor, Xin Fang, Stratis Ioannidis, and Miriam Leeser. Garbled circuits in the cloud using fpga enabled nodes. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2019.

[72] Nigel P. Smart. Bristol format. Web page, 2023.

[73] Python Software Foundation. Abstract syntax trees. `https://docs.python.org/3/library/ast.html`, 2021. Accessed: May 10, 2023.

[74] Green Tree Snakes Contributors. Green tree snakes documentation. `https://greentreesnakes.readthedocs.io/en/latest/index.html`, 2021.

[75] Peter M Kogge. A new ripple-through carry lookahead circuit. In *Proceedings of the 1973 IEEE International Symposium on Computer Arithmetic*, pages 140–144. IEEE, 1973.

[76] Richard P Brent and Hsiang-Tsung Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, 1982.

[77] Christopher Gentry. A fully secure oblivious ram. *Journal of Cryptology*, 22(1):1–40, 2009.

[78] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, Washington, D.C., August 2015. USENIX Association.

[79] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[80] Andrew Yao. Yao's millionaires' problem. Wikipedia, 1982. `https://en.wikipedia.org/wiki/Yao%27s_Millionaires%27_problem`.

[81] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.