Advancing architecture optimizations with Bespoke Analysis and Machine Learning

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

Subhash Sethumurugan

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

John M Sartori

February, 2023

Acknowledgements

I sincerely thank my academic advisor Prof. John Sartori for his guidance throughout my PhD. I am grateful for his enormous support all these years. He was always available for any help or advice both professionally and personally. He constantly encouraged me to pursue any research idea that interested me without any restrictions. I am truly thankful for the opportunity and freedom. I would also like to express my gratitude to the National Science Foundation for supporting my research through the NSF grants. I would like to thank my internship mentor and collaborator, Jieming Yin, for guiding me during an important phase of my PhD. I would also like to thank my other collaborators, Kangjie Lu and Henry Duwe, without whom, my research work would not be complete.

My PhD journey would not have begun if not for the guidance and counseling from my labmate and collaborator, Hari Cherupalli. He was instrumental in all my research endeavors and had spent hours working with me on my projects, even after he had graduated from the University. I would like to convey my special thanks to Shashank Hegde, who can be considered a partner in my PhD, for having worked with me for several years. In addition to learning a lot from him, I believe my problem-solving abilities have been enhanced by our discussions and debates. I would also like to thank all my other labmates - Himanshoo Sahoo, Shaman Narayanan, Nishanth Somashekara Murthy and Tariq Azmy, for all their support.

I would like to express my gratitude to all my university friends - Karthik Srinivasan, Ramya RamaSubramanian, Abhinav Vishwanathan Sambasivan, Vaishnavi Govindarajan, Venkat Ram Subramanian, Deepak Srivatsav Sridharan, Jayasimha Bezawada and many more for making my time in Minneapolis fun and memorable. I would also like to thank my childhood friends - Ashwin, Sharath, Surya, and Nayantara for cheering me up during all the stressful times.

Any of my achievements would not have been possible without the support of my wife and parents. My wife, Sarada Peruboina, is my pillar of strength and confidence. I cannot thank her enough for sharing my life and my responsibilities throughout this period. I thank my parents for encouraging, motivating, and supporting me all throughout my life to realize my goals. I would also like to thank my brother - Akash Subramaniam Sethumurugan, for his constant support.

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. Inspired by the ability of ML-based tools to automate architecture optimization, we also apply ML-guided architecture design and optimization for other challenging problems. Specifically, we target cache replacement, which has historically been a difficult area to improve performance. Furthermore, improvements have historically been ad hoc and highly based on designer skill and creativity. We show that ML can be used to automate the design of a policy that meets or exceeds the performance of the current state-of-art.

Contents

List of Tables

List of Figures

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 [1,2], wearables [3,4], and IoT applications [5–10], 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 [11–13]. 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 [14] 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 [11,12]. 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 [11–13].

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 [15] maintains *conservative* states at PC-changing instructions. 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. 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.

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 [16]. 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 [13] 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. 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.

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 top few predicted architectures using our generic tool and pick the design with most energy-efficiency.

With the help of Machine Learning, we were able to solve the design space exploration problem efficiently. This motivated us to use ML in other complex problems. ML expedites tedious processes and augments human intelligence to solve problems where heuristic-based solutions have limitations. Because of these benefits of ML, there is a surging interest in applying machine learning (ML) to challenging computer architecture design problems. Building an effective cache replacement policy has been one of the important challenges in computer architecture. There has been considerable work on designing efficient cache replacement algorithms [17–22]. The design of novel policies has historically been based on the designer's skill and creativity to derive insights from common cache access patterns and use them to develop a cost-effective cache replacement policy. Firstly, it takes enormous skill and effort to analyze an access pattern and come up with valuable insights. Secondly, new applications emerge every day and existing applications keep evolving, exposing caches to new access patterns. The cache

architecture needs to adapt to the ever-changing demands. Therefore, we need an automated way of generating insights and developing cache replacement policies. In this work, we develop a tool that analyzes a set of applications and learns a near-optimal replacement policy using ML. The designer can then derive a cost-effective replacement policy using the insights from the learned policy. In this work, we show one way of developing a cost-effective cache replacement policy that meets or beats the performance of current state of art policies.

This dissertation is organized as follows. Chapter 2 provides the background for co-analysis techniques used to devise application-specific hardware optimizations. Chapter 3 introduces application-based software constraints and discusses the means and impact of using them for application-specific hardware optimizations. Chapter 4 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 extends the use of Machine Learning in architecture optimization by showing how a cost-effective cache replacement policy can be built using machine learning that beats the current state of art. 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 [1–10] provides the opportunity to make application-specific optimizations on the processor used in such systems. A recent line of work [11–13] 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 [14] 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 [11–13] 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 [11–13] 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 [15] 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, [15] 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

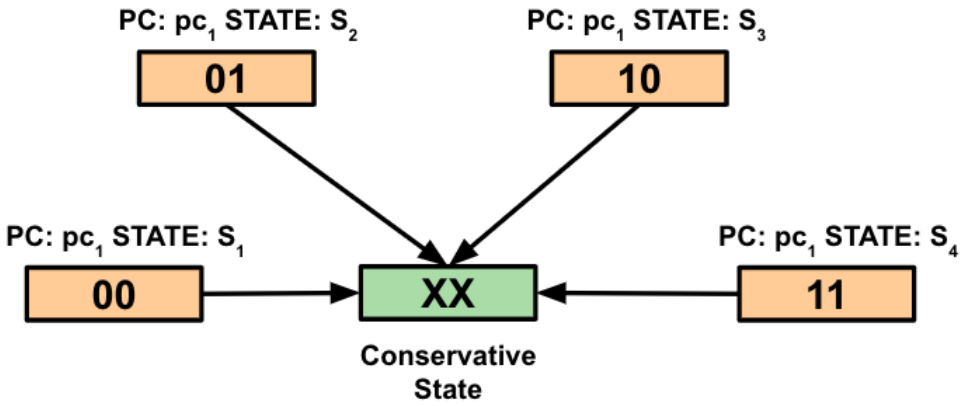


Figure 2.1: Example of conservative state representing register values at different execution states for the same PC.

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.

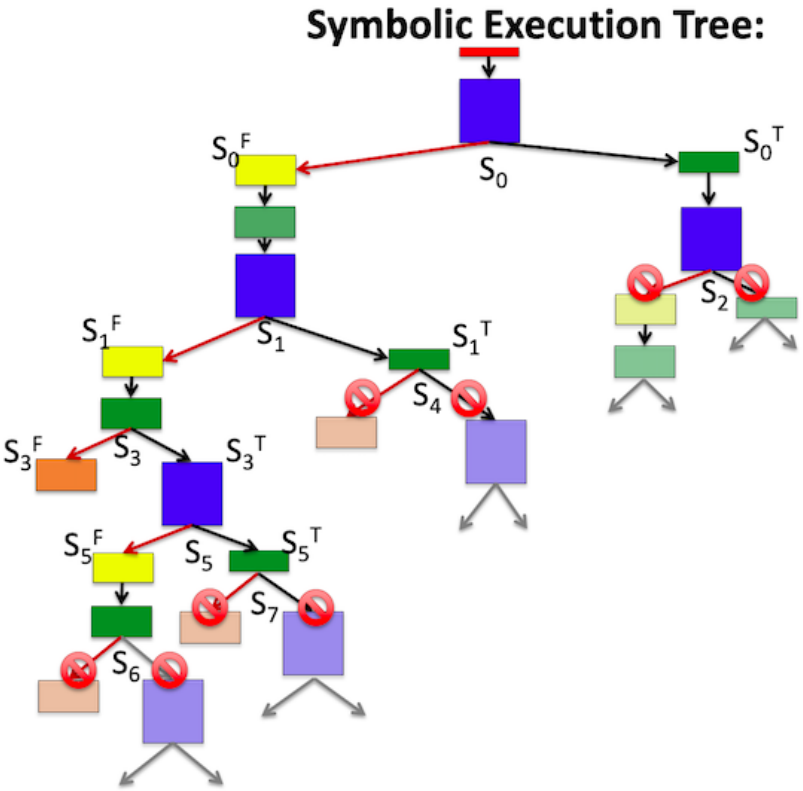
To get the gate activity of the example code, the symbolic simulation replaces the external value (represented by `r14` (`q`)) with Xs. Figure 3.3 shows the execution tree of conservative state based symbolic simulation and the values of two registers `r13` and `r5` at various states that the processor 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.

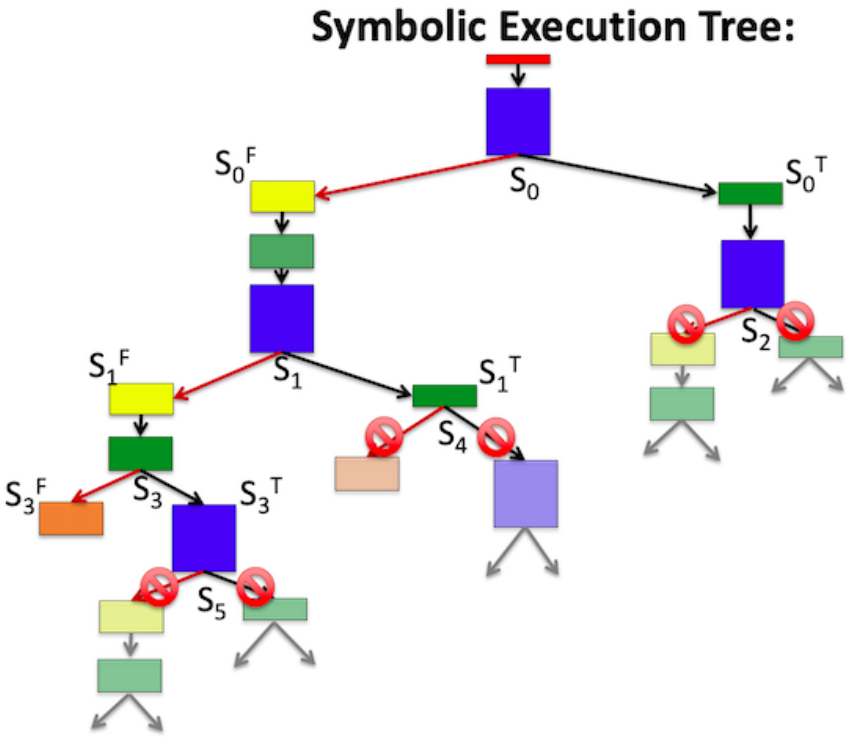
In our work, constrained conservative state symbolic hardware-software co-analysis, we translate constraints on variables at the software level to constraints on memory elements in the processor-memory system. In this example, since $0 \leq$ `r5` < 17, we constrain the value of `r5` to 16'b00000000000XXXXX, 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	**0b000000000000xxxx**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b000000000000xxxx**
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	**0b000000000000xxxxx**
S_2	6	0b0000000000000010	0b0000000000001111
		0bXXXXXXXXXXXXXXXX	**0b000000000000xxxxx**
S_3	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	**0b000000000000xxxxx**
S_4	9	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	**0b000000000000xxxxx**
S_5	6	0bXXXXXXXXXXXXXXXX	0bXXXXXXXXXXXXXXXX
		0bXXXXXXXXXXXXXXXX	0b000000000000xxxxx

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.

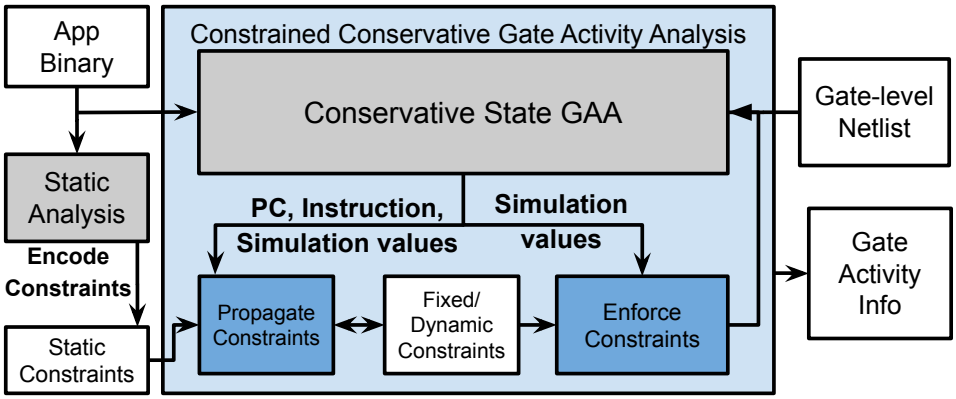


Figure 3.5: Methodology for CCS

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).

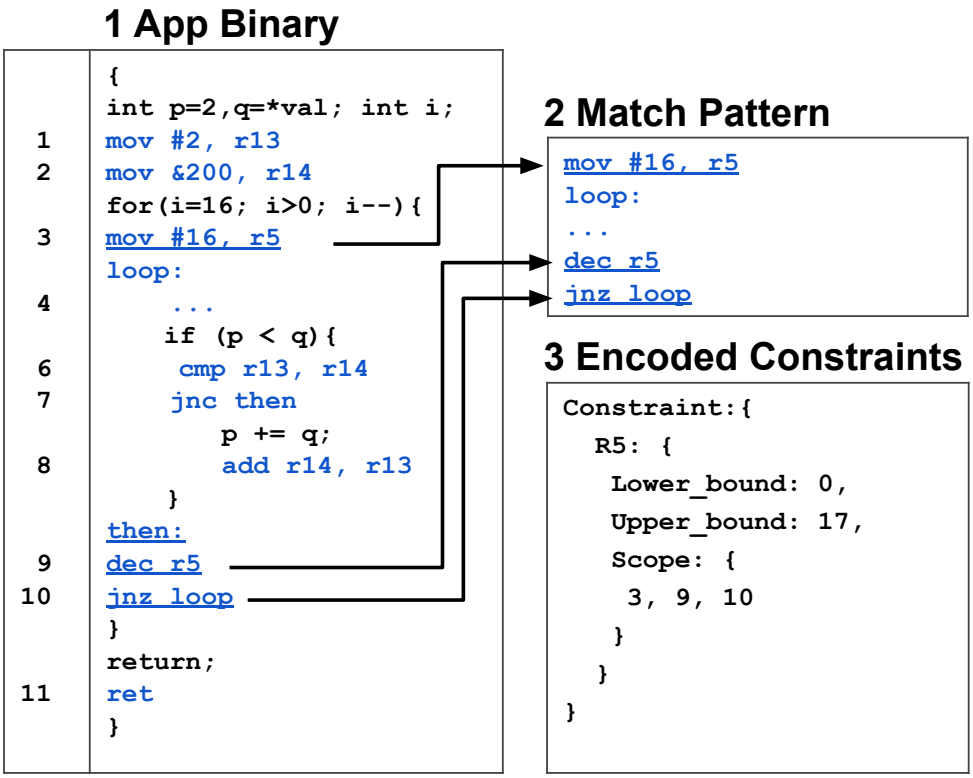


Figure 3.6: Example of constraint encoding during static analysis of the application binary.

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 [23–25]. 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 $2 \leq$ `r13` < 3, representing its constant value. Consider another instruction, `add r5, r13`, with constraints on `r5` as $1 \leq$ `r5` < 17 and on `r13` as $2 \leq$ `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: $18 \leq$ `r13` < 19. However, if the value of `r5` is 16'bXXXXX, the constraint on `r13` is updated to $3 \leq$ `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 i^{th} encounter of $\boldsymbol{p}$. I.e., for all states before

Table 3.1: Benchmarks

Embedded Sensor Benchmarks [29]
mult, binSearch, div, inSort, tea8, rle, tHold, intAVG, intFilt
EEMBC Embedded Benchmarks [30]
AutoCorr, convEn, FFT, Viterbi
Complex Benchmarks
MergeSort , graph500 [31], 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 [26] processor, synthesized, placed and routed in TSMC 65GP (65nm) technology using Synopsys Design Compiler [27] and Cadence EDI System [28]. 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 [29], all EEMBC benchmarks [30] 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 [15] (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 [24] presents an algorithm that detects infinite loops in program using symbolic execution based static analysis. The work in [23] 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. [23] elaborates in detail on some of the applications of static loop analysis. The work in [25] 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 [11–13] 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 [11] 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 [32] 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) [33]. 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

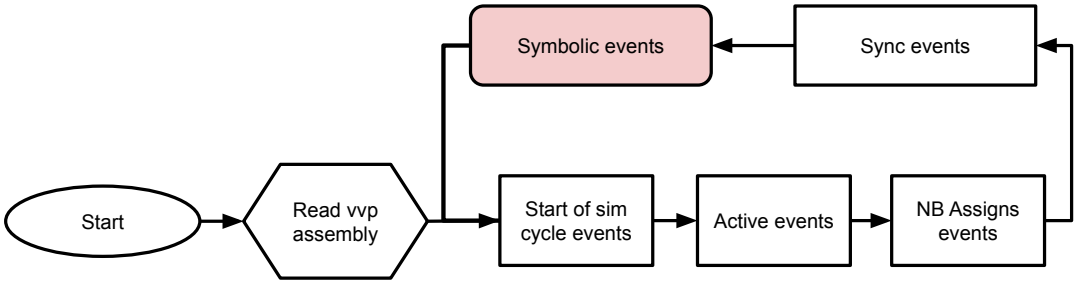


Figure 4.1: We add a new type of event to capture 'symbolic events' in iverilog's event queue. This enables us to monitor control signals for X and halt the simulation when necessary. The VVP engine is a part of iverilog source that executes an iverilog compiled assembly code that is generated from the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

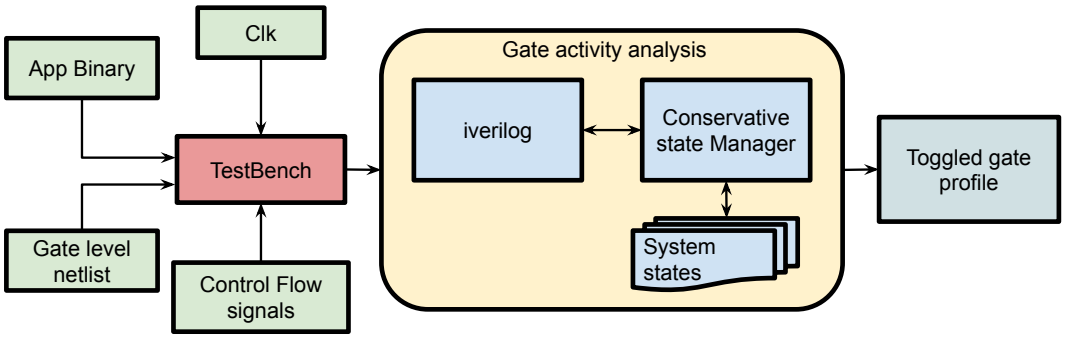


Figure 4.2: Our design-agnostic symbolic co-analysis tool is built on top of iverilog to allow hardware-software co-analysis of any digital design.

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) [34] 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
```

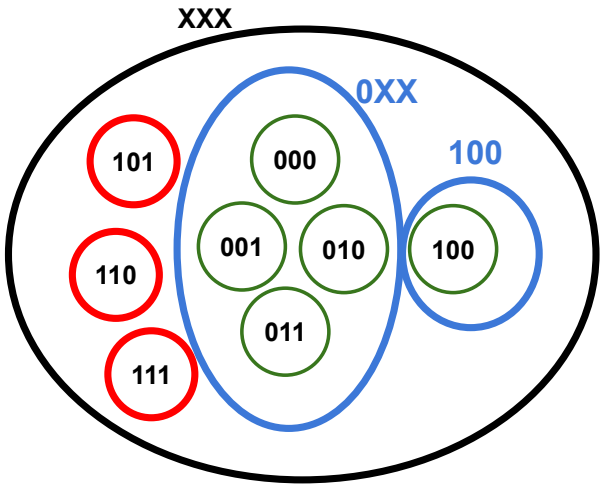
simulation from a previously halted state.

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 [13]. 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

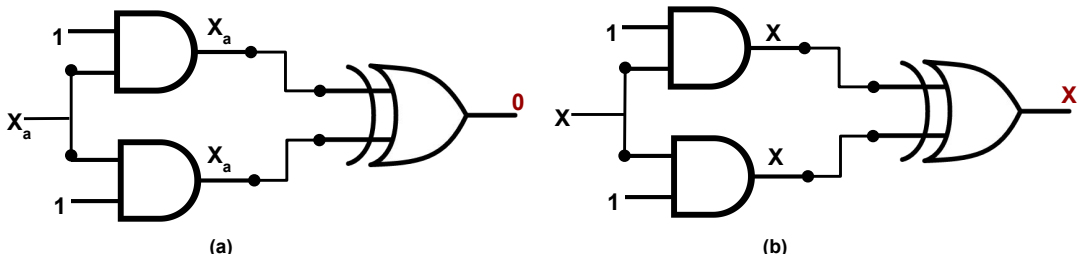


Figure 4.4: Our symbolic tool allows rules for symbol propagation to be customized. The left sub-figure shows a case where circuit inputs are propagated as separate symbolic values, while the right sub-figure shows a case where the symbolic values carry no identifying information and thus cannot be distinguished.

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 [**?**]. 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 [35]. 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 [26] – an open-source version of one of the most popular ULP processors [36, 37], a custom implementation of an open-source 32-bit MIPS processor – bm32 [38] – and DarkRISCV SoC [39], a RISCV implementation that implements the RV32e ISA [40] 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 [27] and Cadence EDI System [28].

Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using our symbolic simulation tool. Table 5.4 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 [34] written in C++ and a Conservative State Manager written in Perl. The CSM uses the conservative state approach used in prior work [13].

Benchmarks are chosen to be representative of emerging ULP application domains such as wearables, internet of things, and sensor networks [29]. 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 Table 5.4 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 [13].

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 Table 5.4. 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 implemented in the designs and how that affects conditional jumps in an application. In

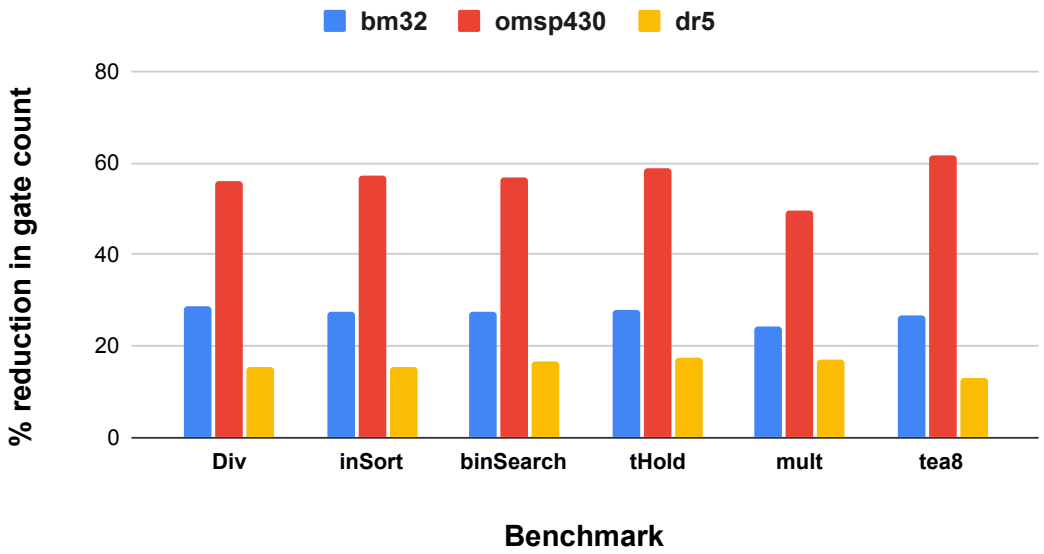


Figure 4.5: Benchmarks run on MSP430 processor have a higher reduction in exercisable gate count compared to MIPS and RISCV processors because of the presence of unused peripherals in MSP430.

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 means that openMSP430 is able to converge faster, while for bm32 and dr5, several

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

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.

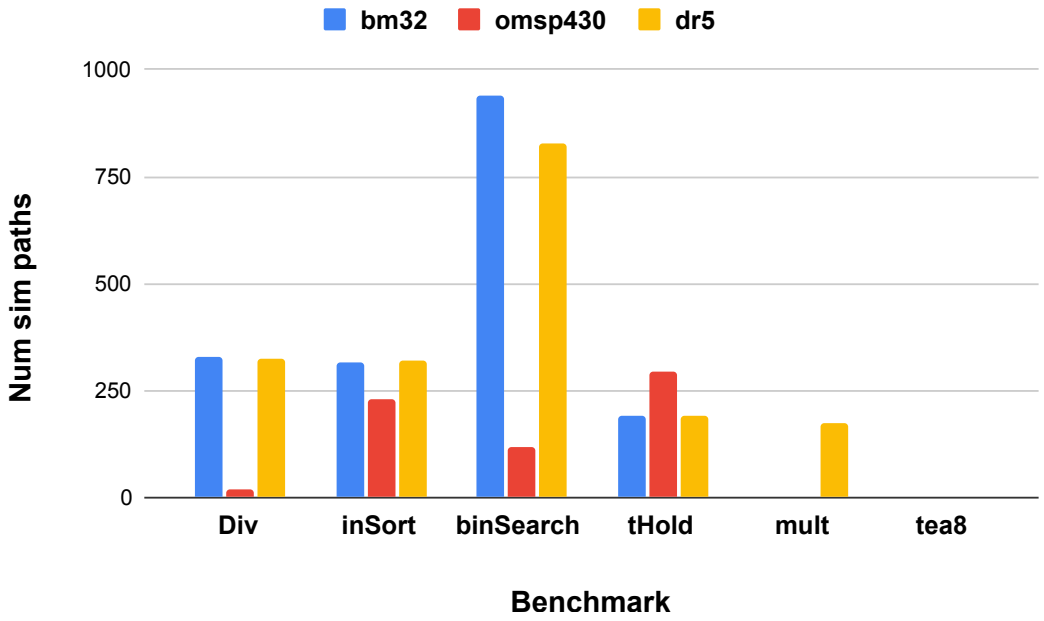


Figure 4.6: Benchmarks run on MIPS and RISCV processors have a higher number of simulated paths because a 16-bit register is used to indicate branch conditions, whereas in MSP430, a 1-bit register is used, resulting in fewer conservative states.

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 [35], to performing application-specific optimizations that reduce power and energy without sacrificing performance or functionality [11, 12, 15, 41], to automatically generating application-specific bespoke processors for ultra-low-power embedded systems [13]. 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 [42] 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 [33]. 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 [39]. 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.

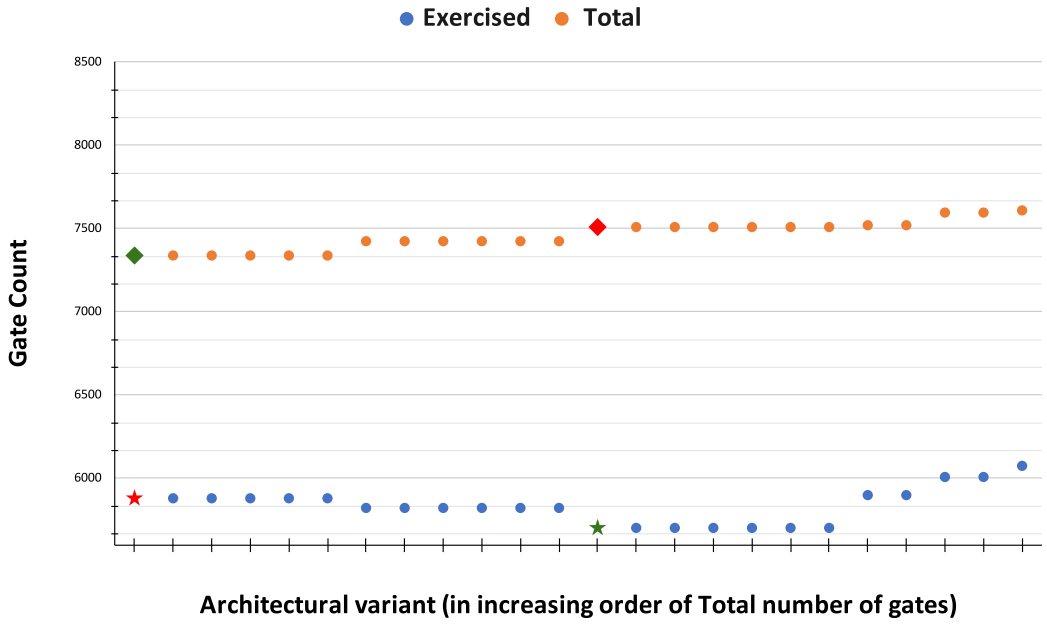


Figure 5.1: This plot shows the total gate count for various architectural variants of the darkriscv processor both before and after bespoke customization for the tea8 application. The optimal processor variant before customization (green diamond) is different than the optimal bespoke processor variant 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 [43] – 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 [44]. 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 [26], MIPS-based bm32 [38], and RISC-V-based darkriscv [39].[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.

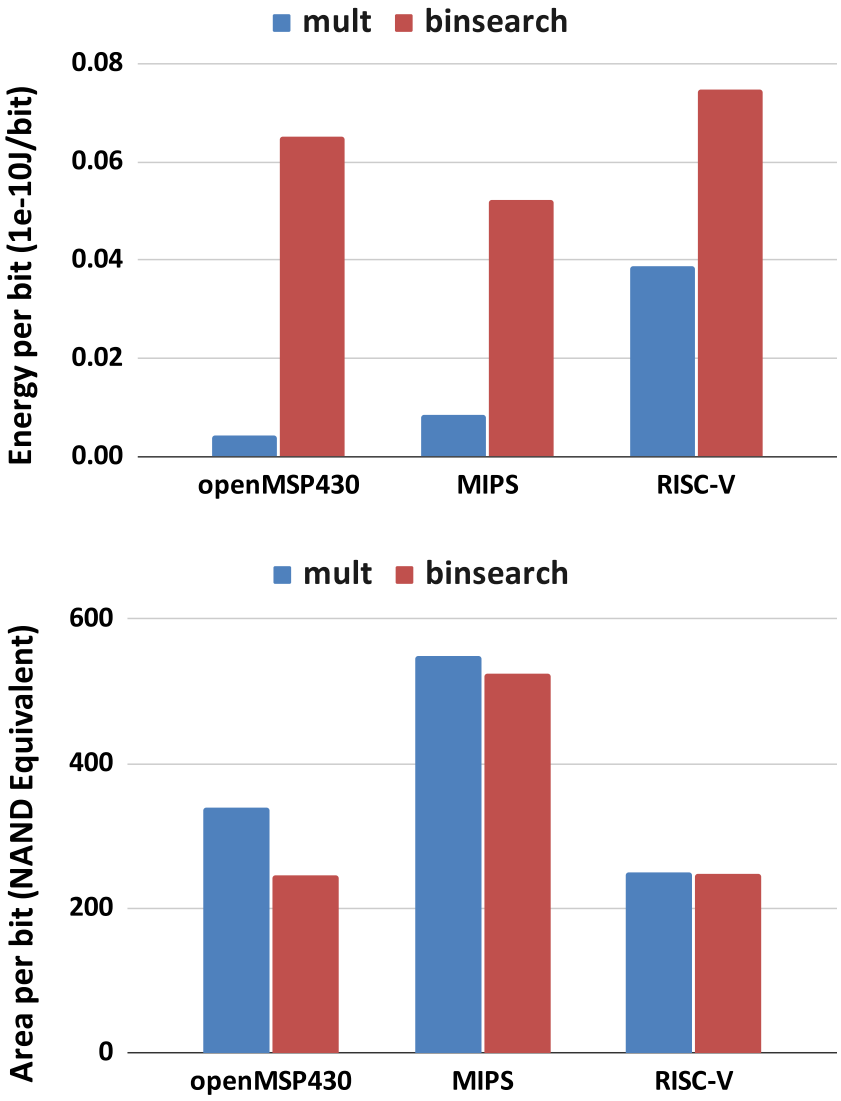


Figure 5.2: The plots compare per-bit energy consumption (top) and area (bottom) for bespoke processors tailored for mult and binsearch applications, starting from three distinct processor architectures – MSP430, MIPS, and RISC-V. The MSP430-based bespoke processor has the lowest per-bit energy consumption 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.

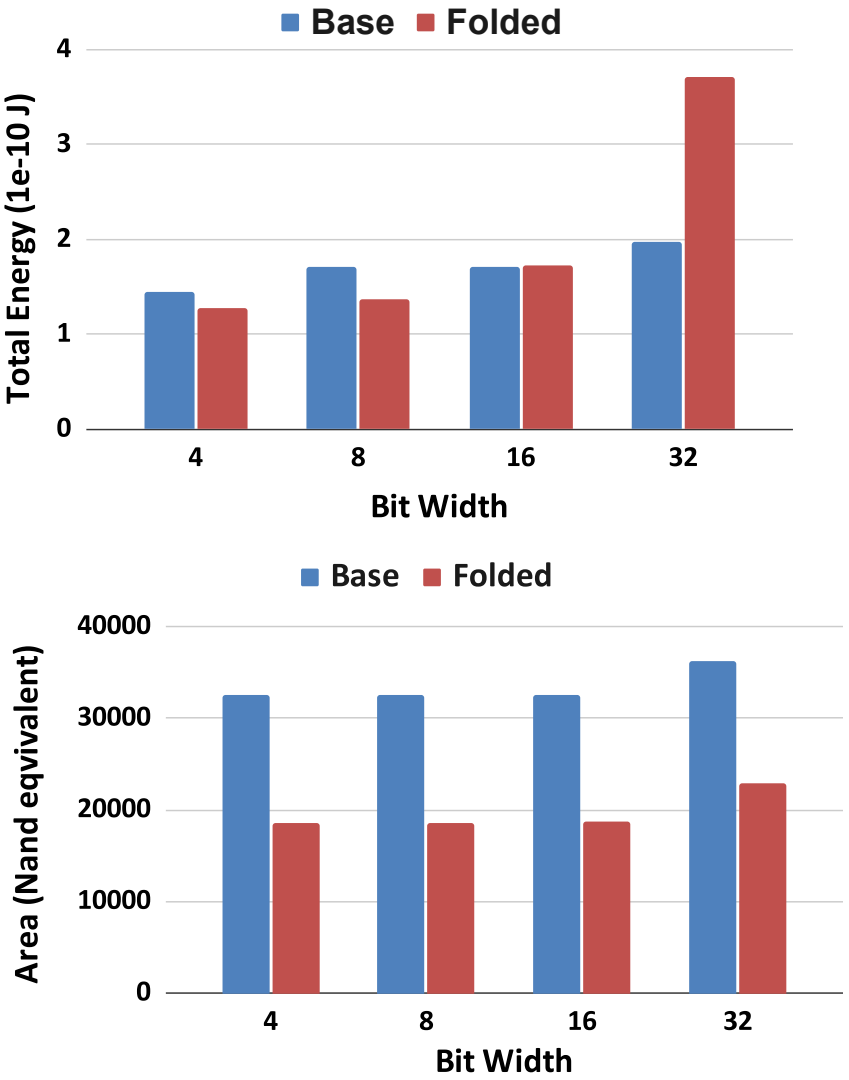


Figure 5.3: The plots compare energy consumption (top) and area (bottom) for bespoke accelerators tailored for applications with different input bit precision, starting from folded and un-folded architectural variants of a 32-bit DCT filter DSP accelerator. The x-axis represents input signal bit width, 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

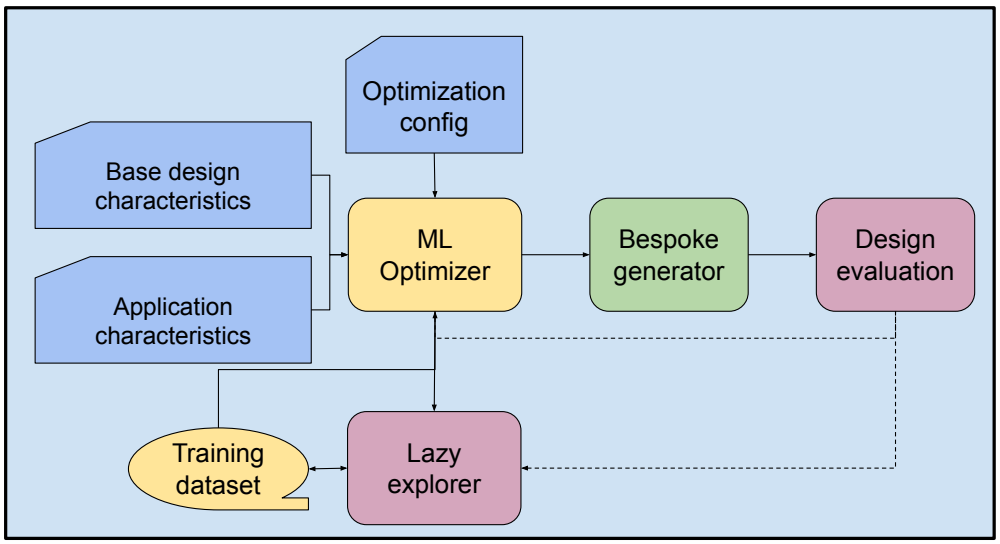


Figure 5.4: Our machine learning model for selecting an architectural configuration from which to generate a bespoke processor uses architecture features of the baseline design and application characteristics to predict metric values for each architectural configuration in the architectural parameter 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.

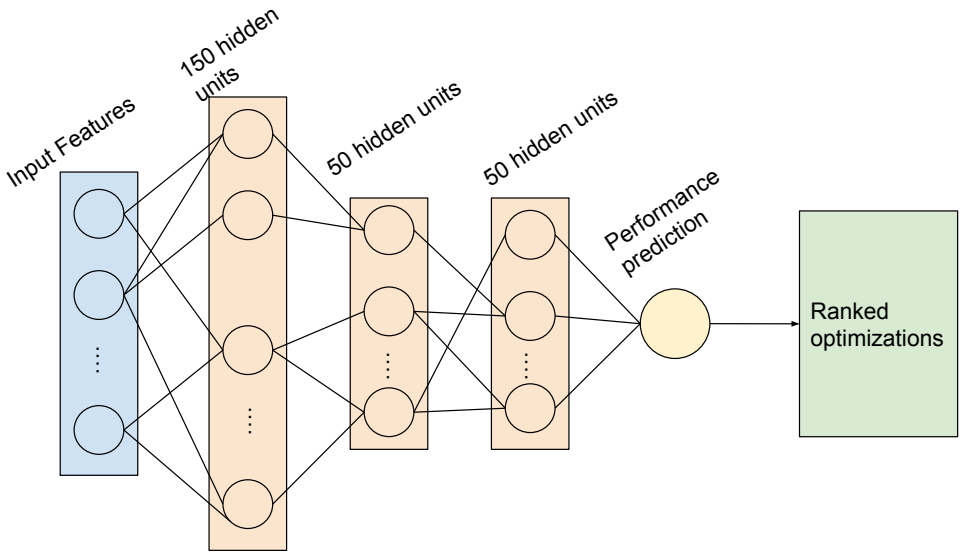


Figure 5.5: We use a neural network that predicts different desired metrics. There is a slight variation in the models that predict area and energy metrics. The model that predicts energy uses the tanh activation function, while the model that predicts area uses ReLU activation.

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 6.4).

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 [26] – an open-source version of the popular ultra-low-power processor MSP430, bm32 [38] – a custom implementation of an open-source 32-bit MIPS processor, and DarkRISCV SoC [39] – a RISCV processor that implements the RV32e ISA [40] 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 [45, 46]. All designs were synthesized using TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [27]. For each design, we explored an architectural parameter space with 58 different architectural variants. The variants are produced by using the Synopsys DesignWare Library [44] 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.

Table 5.4 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 [29]. 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 of the

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

design using the tool described in Chapter 4. For model development, we used machine learning models that are available in the Scikit-Learn module available in Python [47].

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 Table 5.4. 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 registers in the design, the number of bussed registers (registers that have more than one

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

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.

With our model, we aim to discover the optimal architecture within the top 10% of

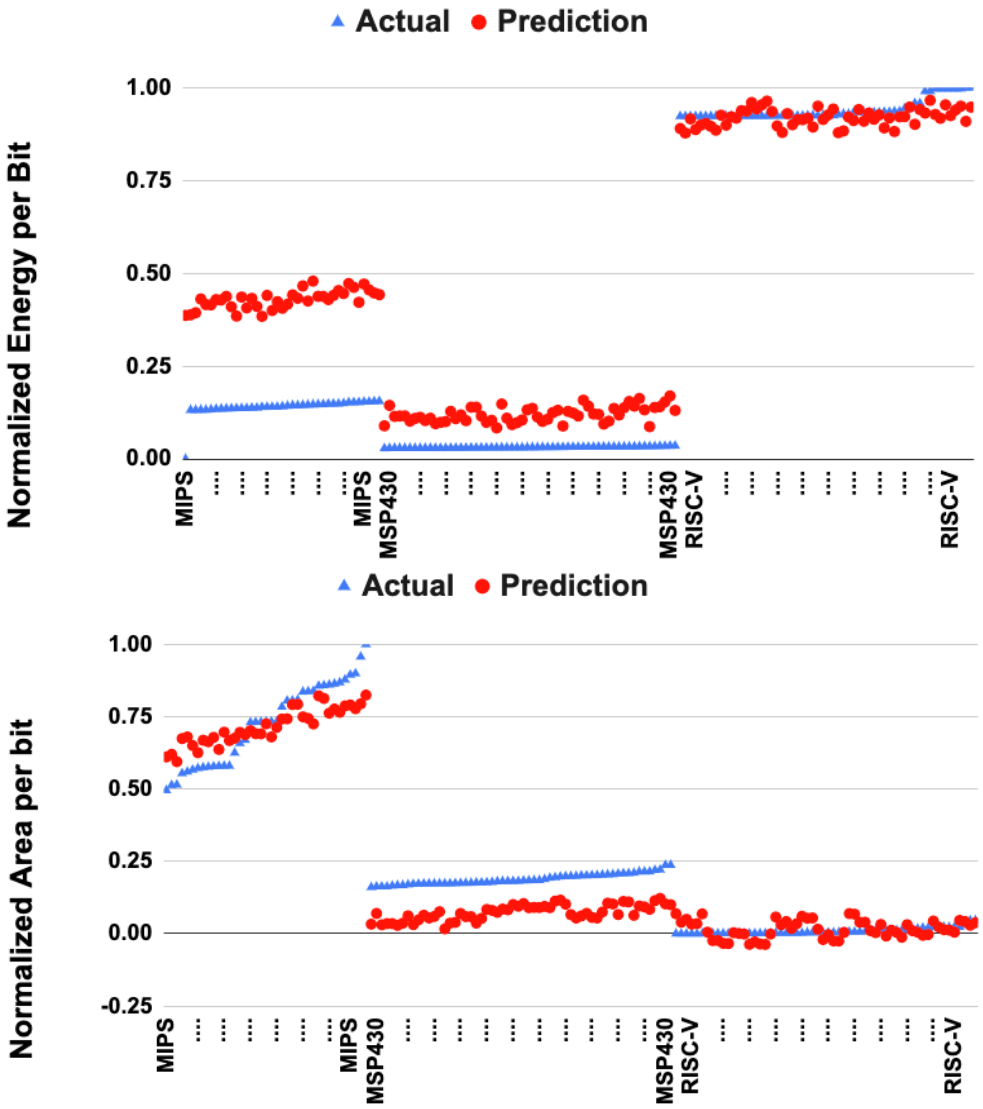


Figure 5.6: This plot shows the normalized energy per bit (top) and normalized area per bit (bottom) predictions of bespoke processors for mult. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend.

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

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.

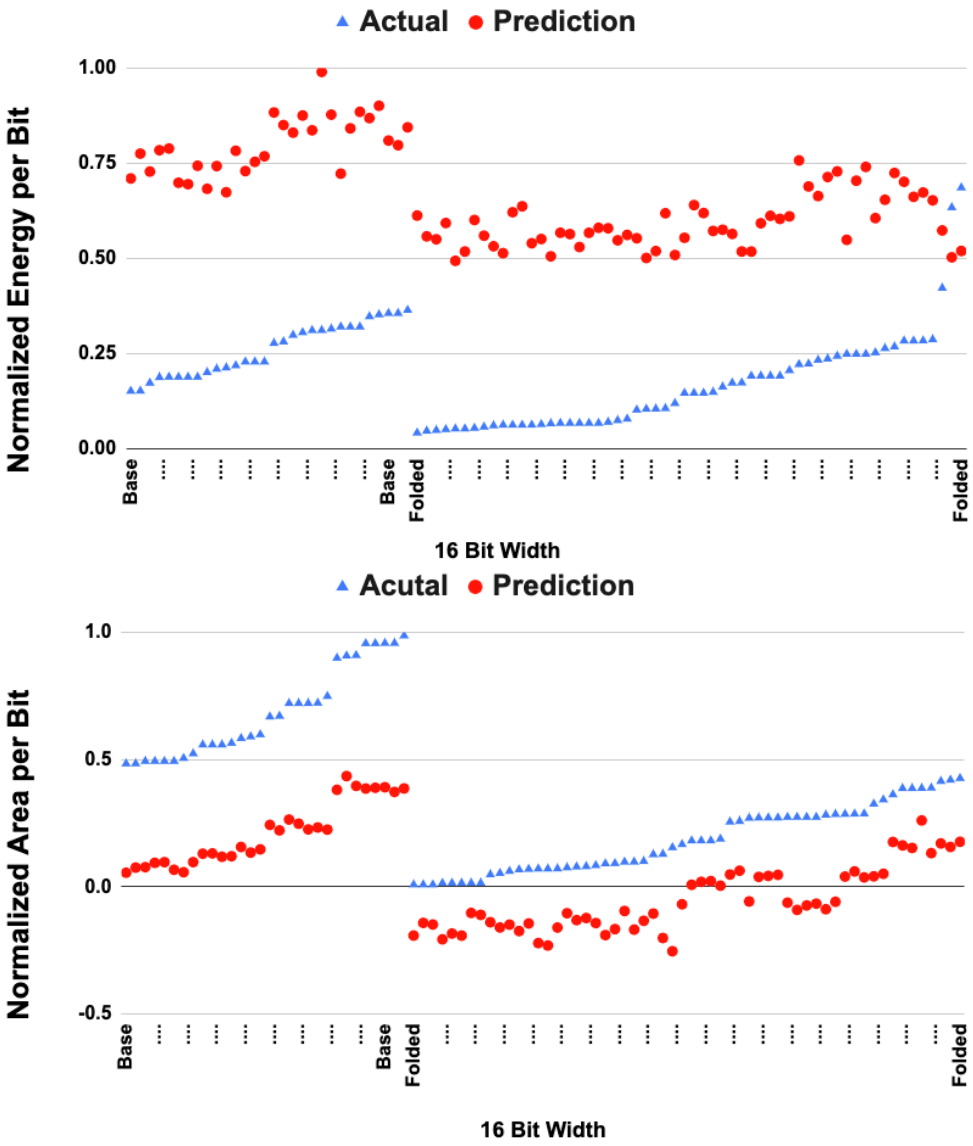


Figure 5.7: This plot shows predicted and actual values for normalized energy per bit (top) and normalized area per bit (bottom) for bespoke DCT accelerators. The x-axis denotes different architectural configurations. The predicted and actual metric values follow a similar trend, indicating that our 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 [48] discuss microarchitecture optimization of the Intel Pentium Pro processor by tuning various microarchitectural parameters, such as pipeline length, cache size, and load store ports. [49] 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 [50] 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 [51] and [52] 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 [53] and [54] explore techniques for design space exploration in a single ISA heterogeneous chip multiprocessor setting. Going beyond multi-core processors, authors in [55] 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 [56], 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 [57] and [58] 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 [59] 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 [60] and Siemens Catapult [61] 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 motivated us to apply machine learning to other complex computer architecture problems. In the next chapter, we will discuss how we used machine learning to develop a cost-effective cache replacement policy.

Chapter 6

Designing Cost-Effective Cache Replacement Policy Using Machine Learning

In the previous chapter, we used Machine Learning to efficiently handle the architecture exploration problem. Inspired by the ability of ML techniques to explore an expansive architectural design space efficiently and effectively, handle complicated non-linear interactions between architecture features, and deliver near-optimal solutions, we use ML to automate and enhance the architecture design process. Historically, computer architecture/system designs are carried out based on expert intuitions and heuristics [62]. However, this approach is not scalable considering the increasing complexity of modern systems. In this chapter, we discuss a methodology where ML is used to gain new insights that help design new architectures and automate the system design process. In our work, we used ML to design a cost-effective cache replacement policy.

6.1 Motivation

Caches are an important component in modern processors. The effectiveness of a cache is largely influenced by its replacement policy. An efficient cache replacement policy

Table 6.1: Hardware overhead for different replacement policies in a 16-way 2 MB cache

Policy	Uses PC	Overhead
LRU	No	16KB
DRRIP [63]	No	8KB
KPC [69]	No	8.57KB
MPPPB [64]	Yes	28KB
SHiP [65]	Yes	14KB
SHiP++ [66]	Yes	20KB
Hawkeye [67]	Yes	28KB
Glider [68]	Yes	61.6KB
RLR (this work)	**No**	**16.75KB**

can effectively reduce off-chip bandwidth utilization and improve overall system performance. There is a large body of prior work on cache replacement policies; however, designing cost-effective cache replacement policies is still challenging for chip designers, especially under stringent hardware constraints.

Cost-effectiveness is becoming increasingly important, as Moore's Law has slowed down and Dennard scaling has ended. A cost-effective cache replacement policy should be able to reduce misses-per-kilo-instructions (MPKI) without introducing significant hardware modification and storage overhead. Commonly used Least Recently Used (LRU) and Re-Reference Interval Prediction (RRIP) polices [63] incur minor hardware overhead for storing the recency bits or re-reference counters. However, these static heuristic-driven policies are only effective for a limited class of cache access patterns. Using program counter (PC) information, state-of-the-art replacement policies are capable of capturing dynamic phase changes in cache access patterns, and they can effectively reduce the MPKI for a wide spectrum of workloads [64–68]. belady compares the last-level cache hit rate among different replacement policies, in which Belady is the theoretical optimal. Unsurprisingly, the PC-based policies (SHiP, SHiP++, and Hawkeye) outperform non-PC-based policies (LRU and DRRIP) in almost all benchmarks.

Unfortunately, incorporating PC into the replacement policy not only requires additional storage overhead but also involves significant modifications to the processor's control and data path. Accessing PC at the LLC requires propagating PC through all levels of the cache hierarchy, including widening the data path, modifying cache architecture to store PC, adding extra storage for PC in the Issue Queue, Reorder Buffer

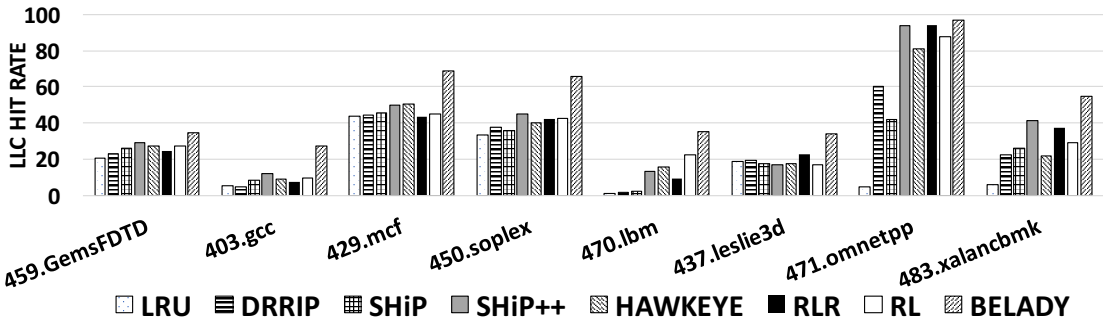


Figure 6.1: LLC hit rate comparison (Belady is the theoretical optimal).

(ROB) and Load/Store Queue (LSQ), and more [69].

More costly than the overhead of implementing PC-based policies is the fact that these changes require an overhaul of the entire processor pipeline, which would require significant design and verification overheads. Given the extensive modifications required, chip manufacturers have thus far been unwilling to implement PC-based replacement policies, favoring instead incremental design enhancements with similar performance/overhead tradeoffs that do not require significant redesign and verification. For example, for the same memory overhead of implementing SHiP++ in LLC, a designer could implement DRRIP in LLC and also increase L1 size by 12KB. Given that increasing L1 size may provide similar or even better benefits without requiring significant redesign and verification, processor manufacturers have as yet not implemented PC-based replacement policies. overhead summarizes the hardware overhead of different cache replacement policies.

How can we improve the cache replacement policy without using PC? We realized this is a difficult challenge. To answer this question in a cost-effective way in terms of reducing product development time, we turned to machine learning for help. Machine learning is a useful tool to augment human intelligence and expedite the chip design process, and computer architects have been using ML to advance computer architecture designs. There has already been work on utilizing ML to improve branch predictors [70], memory controllers [71], reuse prediction [72], prefetchers [73], dynamic voltage and frequency scaling management for network-on-chip (NoC) [74, 75], and NoC arbitration policy [76, 77].

In this work, we explore the capability of ML in designing a cost-effective cache replacement policy. Specifically, we use reinforcement learning (RL) to learn a last-level cache (LLC) replacement policy. The RL algorithm takes into consideration a collection of features that can be easily obtained at the LLC without modifying the processor's control and data path. After successfully learning a replacement policy that achieves good performance, we analyze the learned policy by applying domain knowledge, with the goals of distilling useful information, verifying the important features, understanding the relative importance of each feature, and gaining insights into how these features interact. Guided by ML, we frame a cost-effective cache replacement policy – Reinforcement Learned Replacement (RLR). Overall, RLR does not require heavy modification to the CPU microarchitecture and outperforms LRU and DRRIP for most evaluated benchmarks. belady compares LLC hit rate for several replacement policies, including a policy learned by RL and our static adaptation (RLR) based on RL.[1] The performance of the RL policy is better than LRU, DRRIP, SHiP, and Hawkeye in most benchmarks, while it is marginally lower than SHiP++. RL performance can be improved by including PC-based features in the feature set, but one goal of our work is to design a cost-effective replacement policy that does not rely on PC at the LLC. The performance of RLR shows the effectiveness of the insights learned from RL.

6.2 Machine Learning-Aided Architecture Exploration

Reinforcement Learning is a machine learning paradigm in which an agent tries to navigate through an environment by choosing an action from a set of allowed actions [78]. Using the suggested action, the environment moves from the current state to the next state and meanwhile generates a reward as a feedback to the agent. The agent trains itself to maximize cumulative reward. In this process, the agent learns a policy that selects the optimal action in a given state. One way to keep track of the optimal action for a given state is to maintain a table for all state-action pairs. However, it could be infeasible to implement such a table when the state and action space is large. In such a scenario, a neural network can be used as a function approximator in lieu of a table.

[1]Due to the computational complexity of running RL simulations and the need to look at future accesses, RL and BELADY simulations are run in a python-based simulator; the rest of the policies use ChampSim. Further details are explained in Section 6.2.

RL has the potential to learn a theoretical optimal policy, given that the effects of actions are Markovian [79]. Because RL has the ability to adapt to dynamic changes in the environment and handle the non-trivial consequences of chosen actions, it is a good fit for the cache replacement problem. We pose cache replacement as a Markov Decision Process (MDP), where an agent makes replacement decisions. Given a cache state, the replacement decision made by the agent moves the cache to a new state. The agent is assigned a reward based on how close the replacement decision is to BELADY (optimal). In our framework, we train a neural network using RL algorithms to learn a replacement policy. Although the learned policy can be efficient, we do *not* want to build a neural network in hardware, due to power, area, and timing constraints. Instead, we analyze the neural network and use the insights gained from the neural network to derive a replacement algorithm that is feasible to implement in hardware. In this section, we describe our simulation framework and architecture exploration flow in detail.

6.2.1 RL-based Simulation Framework

At high level, our simulation framework consists of two parts: trace generation and RL training. We use ChampSim [80] from the 2^{nd} Cache Replacement Championship (CRC2) to generate LLC access traces. The trace file comprises a record of $\langle$*PC, Access Type, Address*$\rangle$ for each LLC access. Access types include load (LD), request for ownership (RFO), prefetch (PR), and writeback (WB). We use LRU as the default replacement policy to ensure that the generated traces are not biased towards other policies that we compare against.

The trace is fed into a Python-based cache simulator that includes an RL agent to make replacement decisions. The cache simulator uses the same LLC configuration as ChampSim and populates the LLC based on the accessed addresses. Each cache line is associated with a collection of cache states, as described later in this section. The simulation framework is shown in Figure 6.2. On a hit, the cache simulator updates the cache states and moves on to the next access. On a non-compulsory miss, the cache simulator interacts with the agent to make a replacement decision 1. Information regarding the missed access and the accessed set is sent to the agent in the form of a state vector 2. The agent evaluates the state vector and generates an output vector of n elements for an n-way set associative cache 3. Each element in the output vector

Table 6.2: List of features considered by the RL agent

Classification	Feature	Description
Access Information	offset	Lower order 6 bits of accessed address
	preuse	Set accesses since last access to the accessed address
	access type	Type of access (LD, RFO, PR, WB)
Set Information	set number	Set that was accessed
	set accesses	Total number of set accesses
	set accesses since miss	Set accesses since last miss to the set
Cache Line Information	offset	Lower order 6 bits of cache line address
	dirty	Dirty bit of the cache line
	preuse	Set accesses between last two accesses to the cache line
	age since insertion	Set accesses since cache line insertion
	age since last access	Set accesses since last access to the cache line
	last access type	Type of last access to the cache line (LD, RFO, PR, WB)
	LD access count	Number of load accesses to the cache line
	RFO access count	Number of read-for-ownership accesses to the cache line
	PF access count	Number of prefetch accesses to the cache line
	WB access count	Number of write-back accesses to the cache line
	hits since insertion	Number of hits to cache line since its insertion
	recency	Order of cache line access with respect to other cache lines in the set

corresponds to a way in the cache set, and the value represents how beneficial it is (from the agent's perspective) if a certain way is chosen for eviction. The cache simulator then makes a replacement decision based on the output vector generated by the agent; meanwhile, a numerical reward is generated and sent to the agent for further training 4. Below we explain the critical components is detail.

State Vector: LLC state vector contains information required to make a replacement decision. We segregated LLC state into three classes of features: a) access information that describes the current access to the cache; b) set information that describes the set that is being accessed; and c) cache line information that describes each cache line in the set that is being accessed. On every LLC access, statistics of the accessed set are updated. For example, the counter *set access* is incremented on every access to the set. As another example, the counter *set access since miss* is incremented on every hit to

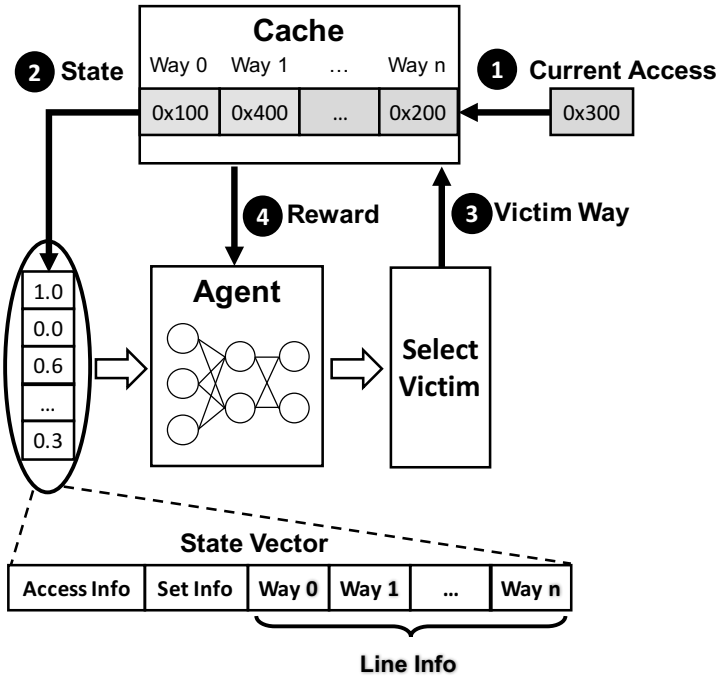


Figure 6.2: Simulation framework overview.

the set and reset to zero on a miss. Similar counters are maintained for every cache line in the set. The cache line counters are reset in the event of an eviction to start counting for the newly inserted cache line. Cache lines are also augmented to store other information such as their recency, last access type, dirty bit and other relevant features. The entire feature list representing LLC state is listed in Table 6.2. Categorical features such as *last access type* are one-hot encoded. Numerical features such as *access count* are normalized by their respective maximum values and represented as a fractional value between 0 and 1. The only exception is the feature *offset*, for which we use a 6-bit binary representation (assuming 64-byte cache lines). For a 16-way set associative LLC, we represent a state vector using 334 floating point values.

Agent: The agent consumes the state vector and generates an output vector of size equal to the set associativity of the LLC. Each value in the output vector indicates the estimated quality of choosing the cache line in the corresponding way as a victim. In this work, we use a neural network to estimate the quality. After extensive exploration on

neural network architecture and hyperparameter tuning, we chose to use a multi-layer perceptron (MLP) with one hidden layer, because it is simple enough for interpretation but performs almost as well as denser networks. We also found that *tanh* activation for the hidden layer and *linear* activation for the output layer yielded better performance than other combinations. The neural network has 334 input neurons, 175 hidden neurons and 16 output neurons (because of the 16-way LLC). On every cache miss, the simulator queries the agent to select a victim. In our simulation framework, there is only one neural network for victim selection for all sets of the LLC. This is similar to following a common replacement policy for all sets. Designers can choose to use multiple agents by training them using different combination of cache sets.

Replacement Decision: The agent returns a vector of n values, one for each cache way (e.g., 16-element vector for a 16-way cache). The replacement decision is made by an ϵ greedy approach [81], in which we choose the victim with the maximum value with a probability of $1 - \epsilon$ and randomly select a victim with a probability of ϵ. Random actions explore new trajectories and expand the search space. In our experiment, we found that an ϵ value of 0.1 yielded better performance than other ϵ values.

Reward: The reward steers the agent towards learning a more optimal replacement policy, so reward function must be chosen carefully. A theoretically optimal replacement policy, such as Belady, replaces the cache line that has the farthest reuse distance among lines in a set. To allow the agent to learn this behavior, a positive reward is returned when the agent makes a good decision and evicts the cache line with the farthest reuse distance. A negative reward is returned when the agent evicts a cache line with a lesser reuse distance than the cache line that is inserted in cache, since the evicted cache line would hit sooner than the inserted cache line if retained in cache. A neutral reward is awarded when any other cache line is evicted. Only the optimal replacement decision is assigned a positive reward, differentiating it from the other decisions and allowing the agent to learn a near-optimal policy faster.

Replay Buffer: For training, we use a technique called experience replay [81]. Each replacement decision is stored as a transaction in a replay memory. A transaction is represented by a tuple of $\langle$*state, action, next state, reward*$\rangle$. A replay memory is a circular buffer with a limited number of entries, and the oldest transaction is overwritten by a new transaction. Instead of using the most recent transaction, the neural network

Table 6.3: Parameters and configuration for RL training

Neural Network	Input Dimension	334
	Output Dimension	16
	Activation	*linear*
	Number of Hidden Layers	1
	Hidden Layer Dimenstion	175
	Activation	*tanh*
	Optimizer	Stochastic Gradient Descent
	Learning Rate	0.01
	Loss Function	Mean-Squared
	Discount Factor	0.1
Replay Memory	Number of entries	500
Benchmarks	459.GemsFDTD,403.gcc,429.mcf,450.soplex, 470.lbm,437.leslie3d,471.omnetpp,483.xalancbmk	

is trained using a batch of randomly sampled transactions from replay memory. Experience replay breaks the similarity of subsequent training samples, which in turn reduces the likelihood of the neural network from being directed into local minima. In addition, experience replay allows the models to learn the past experience multiple times, leading to faster model convergence and reduced training time. In our work, we used a replay buffer consisting of 500 entries.

Training: Table 6.3 shows the summary of RL agent parameters and configuration used for training. For our training, we selected 8 memory intensive SPEC2006 benchmarks, that showed significant IPC improvement for the optimal replacement policy (Belday) compared to the contemporary replacement policies. For all benchmarks, we generated LLC access traces for 100M instructions and trained on them individually. In our training, each epoch constituted simulating all memory accesses in the trace file. The RL agent performance improved significantly in the first three epochs for all the benchmarks. Since each epoch of training took significant time and the agent performance did not improve to a great extent beyond three epochs, we stopped the training process after three epochs.

6.2.2 Insights From Neural Network

Neural networks have the ability to achieve better performance by learning a favorable policy after sufficient training. However, it is likely not cost effective to implement

a neural network in LLC due to significant power and area overheads. To achieve performance similar to that of a neural network while avoiding the associated hardware overhead, we analyze and draw insights from a trained neural network to derive a practically implementable replacement policy.

Applications that show significant difference in LLC hit rates between Belady and LRU replacement policies were chosen for testing. Although state-of-the-art replacement policies perform better than LRU in these applications, there is still a performance gap between these policies and Belady, which provides scope for us to improve our policy. We use reinforcement learning to guide us through the process. We allow a neural network to explore paths unexplored by the contemporary replacement algorithms with the target of closing the performance gap between Belady and other replacement policies. After training, we analyze neural network weights for all selected benchmarks. As stated insection 6.2.1, on a LLC miss, the LLC state representing features of missed access and the accessed set is sent as input to the agent. Then, the agent provides a replacement decision to be followed. To comprehend important features in the LLC's state that affect the agent's decision, we compute the average weight of each individual input layer neuron over all neurons in the hidden layer. For cache line features, we also compute the average across all cache ways in a set. Figure 6.3 shows the heat map of feature weights such that the higher and lower magnitude weights are depicted at different ends of the color spectrum.

Typically, a feature is more important if it has higher magnitude weights (darker color in the heap map). Although the heat map can help identify important features for making good replacement decisions, it is left to us to understand why these features are important and how they impact replacement decisions. Ultimately, we would want to utilize these features and derive a practical replacement policy. Implementing the agent's neural network directly in hardware is unlikely practical.

The first challenge in the process of deriving our own replacement policy is to minimize the search space and focus only on critical features. We use hill-climbing analysis together with machine learning to finalize our target feature set. We started by training the agent with only one feature at a time. After doing this for each individual feature, we select the feature that performs the best. Then we enable this feature with one additional feature and evaluate all such feature pairs. We repeat the process by adding

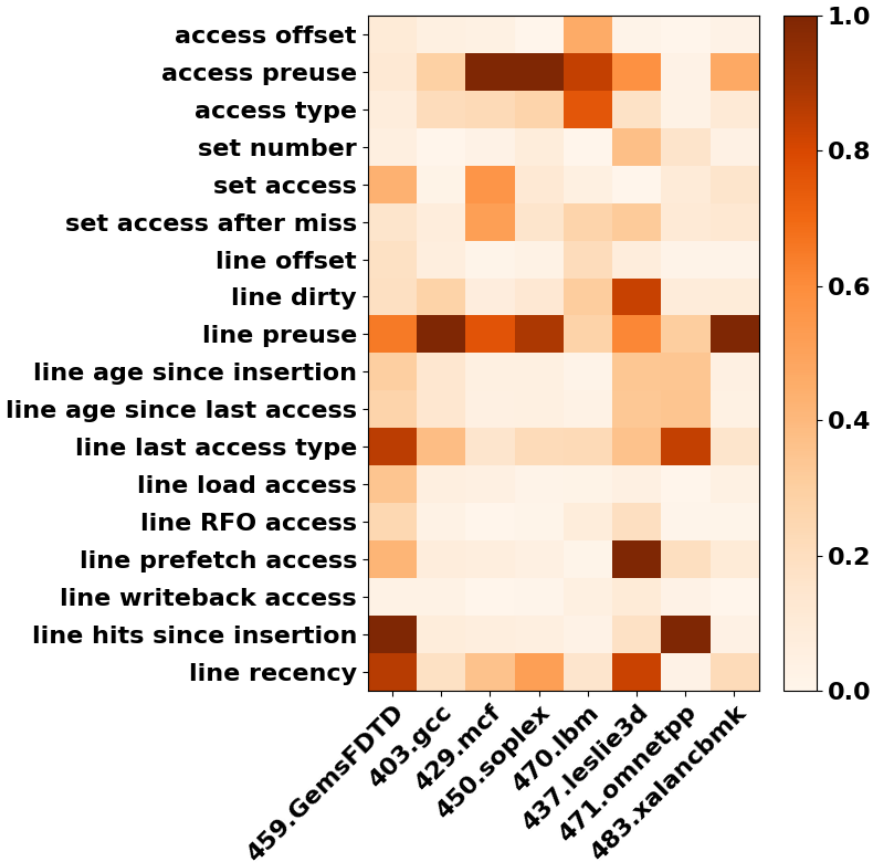


Figure 6.3: Heat map of neural network weights. The y-axis shows features representing LLC state, and the x-axis shows the benchmarks used in the agent simulation. The features with high magnitude of weights are (considering at least three benchmarks) access preuse, line preuse, line last access type, line hits since insertion, and line recency.

one more feature at a time until no further performance improvement is seen. This hill-climbing analysis yields a set of five features.

In this section, we define each of these features and discuss the insights that we derive from them. Later, we design a new policy based on these features that can be implemented in hardware with acceptable overheads.

Preuse Distance

We define preuse distance as the past reuse distance of a cache line. It is computed as the number of set accesses between the last access and the current access of the cache line. Based on the heat map, both access preuse and line preuse features show high magnitude of weights.

Access Preuse is the preuse distance of the cache line that is accessed by the current request. To obtain the preuse distance for every cache access, one must keep a record of all previously accessed addresses, refer to the record when serving a new access, and compute its preuse distance accordingly. Although we implement the record keeping and lookup function in our simulation framework, doing so in hardware can be very costly. As a result, we do not consider this feature for our final policy.

Line Preuse refers to the preuse distance of a cache line. To compute line preuse, we add counters for every cache line. On a set access, the counters of all cache lines in that set are incremented. If the access is a hit, the counter value corresponds to its preuse distance. On a miss, a new cache line is installed. We then reset the counter and start counting the preuse distance of the newly inserted cache line. In Section 6.3, we propose optimizations to reduce the counter overhead.

How does preuse distance contribute to the policy that the agent learned? Recall in Section 6.2.1 that the agent tries to learn the behavior of Belady optimal policy, i.e., replace the cache line with the farthest reuse distance. However, reuse distance is not provided as an input feature. Our conjecture is that preuse distance is related to reuse distance in certain scenarios. During a program execution, the distance between two accesses of an address could be constant. For example, the number of memory accesses in each iteration of a for loop can be the same. In such a scenario, if the same address is accessed in every iteration, its reuse distance will be constant. However, this may not be true in the case of an LLC access because of the filtering effect of private caches. In

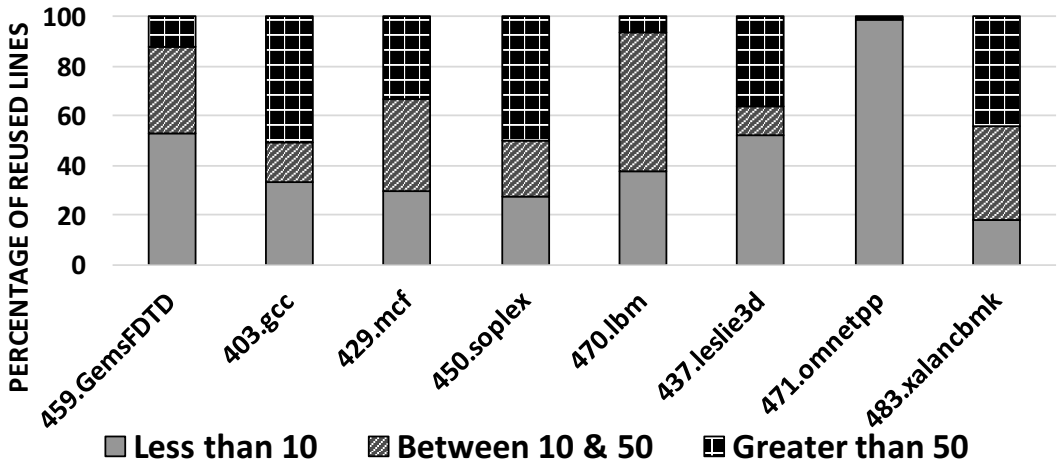


Figure 6.4: Difference between preuse and reuse distance for reused cache lines.

addition, prefetch and writeback accesses can impact reuse distance.

To comprehend the relationship between preuse and reuse distance at LLC, we analyze the difference between preuse and reuse distance for every LLC access. Figure 6.4 shows the percentage of reused cache lines with absolute difference between preuse and reuse distance below 10 (i.e., $|preuse\ distance - reuse\ distance| < 10$), between 10 and 50, and greater than 50. For a significant number of cache lines, we can approximate the reuse distance using preuse distance, as the difference between preuse and reuse distance is less than 10 accesses. For more than 50% of the reused cache lines, the difference between preuse and reuse distance is less than 50 accesses. To allow these cache lines to be reused, we can retain the cache lines for a few more accesses after their preuse distance has been reached. We should also note that Figure 6.4 shows the absolute difference between preuse and reuse. This means that the reuse distance could be smaller than the preuse distance, so some cache lines might be reused before reaching their respective preuse distances.

Line Last Access Type

Last Access type of a cache line is defined as the latest access' type. To understand its significance, Figure 6.5 shows the average victim age for each access type. We accumulate the age since the last access for each victim chosen by the RL agent and

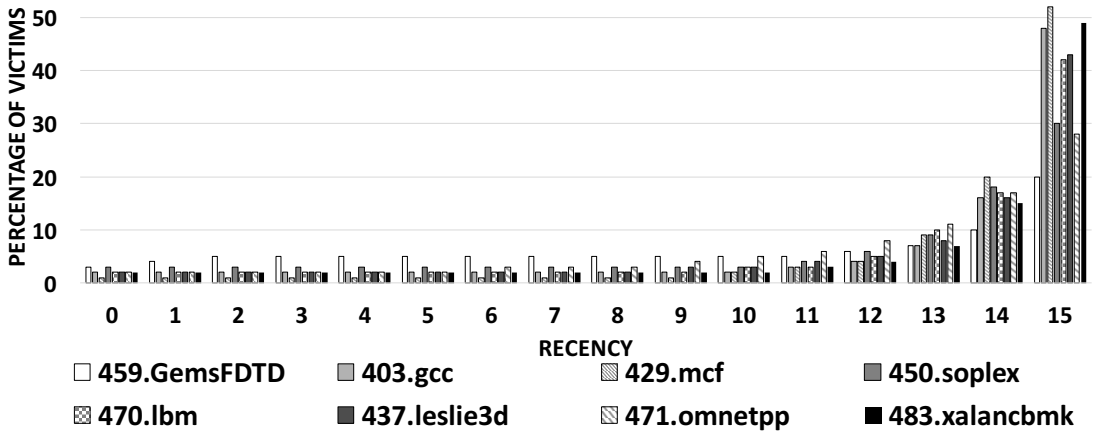


Figure 6.5: Average victim age for each access type.

compute the average for each access type. In almost all benchmarks, prefetch access has the lowest average victim age. This implies that the prefetched cache lines have the lowest cache life time, and the agent prefers to evict them sooner than the cache lines from other access types. However, prefetched cache lines contribute to significant number of demand hits for a few benchmarks, like 459.GemsFDTD, 437.leslie3d, and 429.mcf. Therefore, we infer that the reuse distance of prefetched cache lines is small, and it suffices to have a short cache life time for prefetched cache lines. This ensures that non-reused prefetched cache lines are evicted sooner, allowing other cache lines to be reused.

Line Hits Since Insertion

Hits since insertion tells us how many times a cache line has been accessed since it was brought into the cache. To understand its significance, Figure 6.6 shows the percentage of victim cache lines that are evicted with zero, one, and more than one hits. In all benchmarks, more than 50% of victims have zero hits, and more than 80% of victims have at most one hit. The insight from this analysis is that the agent tends to evict cache lines with fewer hits. When designing a cache replacement policy, we can mimic this behavior by retaining cache lines that have more hits.

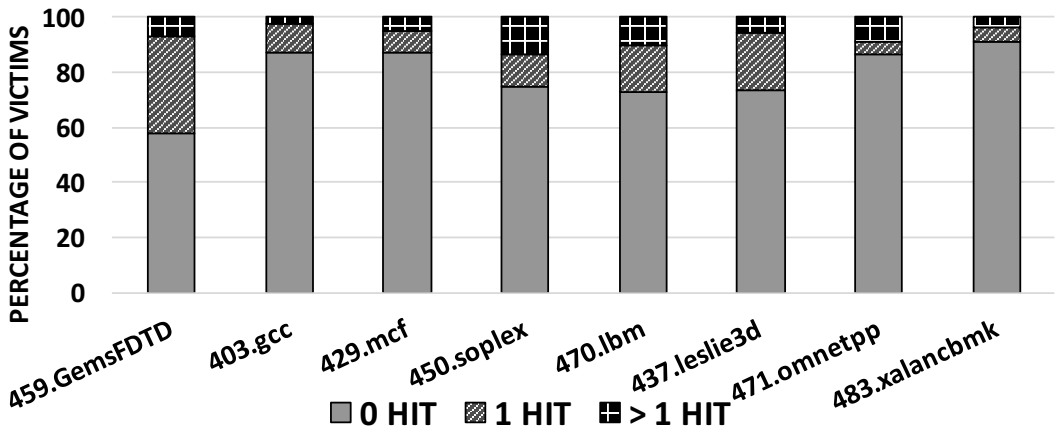


Figure 6.6: Number of hits when a cache line is evicted.

Recency

Recency refers to the relative access order of a cache line in a set. Recency value ranges from zero to $(Set\ Associativity - 1)$; zero indicates the least recently used cache line, and $(Set\ Associativity - 1)$ indicates the most recently used cache line. For example, LRU replacement policy replaces the cache line with recency value 0.

To understand the significance of recency, we plot the percentage of victims evicted by the agent, segregated by recency of the victims, in Figure 6.7. We observe that most evictions occur with cache lines with a high recency value, implying that the agent prefers to evict cache lines that are most recently used. To comprehend this behavior, note that the agent is rewarded positively for evicting cache lines that are either not reused or reused later than the other cache lines in the set. When the agent evicts a cache line with a high recency value, it means that the older cache lines (recency value close to 0) are reused before the newer cache lines (recency value close to 15). For example, when two cache lines in the set have the same reuse distance, the older cache line will be reused before the newer cache line. So, the agent chooses the newer cache line for eviction. Given the high percentage of victims with high recency values in agent simulations, we take the insight that evicting more recently accessed cache lines has a better chance of maximizing demand hits.

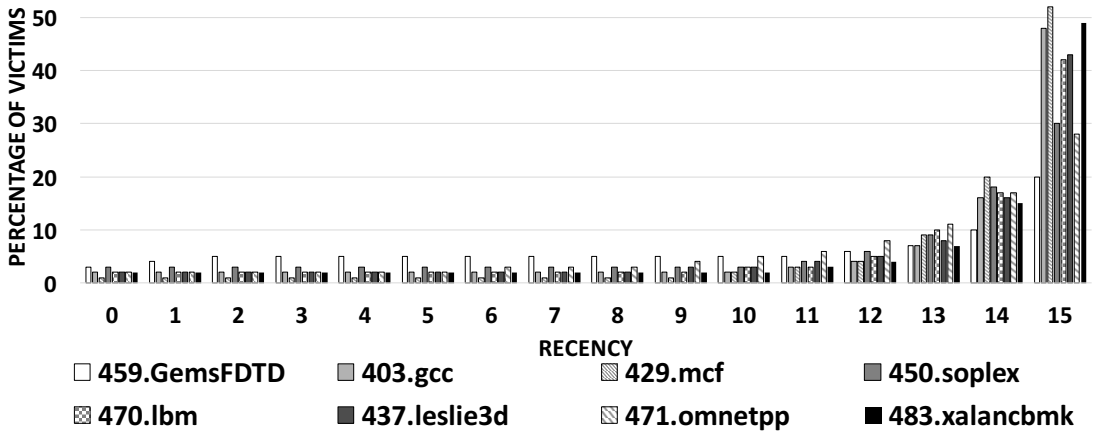


Figure 6.7: Recency for victims in agent simulation.

6.2.3 Benefits Of Deriving Insights Using ML

In this section, we presented one viable way to draw insights from an ML model for cache replacement policies. First, we identify important features by analyzing the weights of the agent neural network. Next, we try to understand the behavior of each feature by looking into its relevant statistics collected from architecture simulations. Through this ML-based analysis, we benefit from the following:

1) *Reducing exploration time*: Several cache replacement policies are built on heuristics identified from common access patterns. New heuristics can be derived through creative and aggressive design of experiments for the cache replacement problem; however, this process is time consuming and limited by a designer's imagination. RL lets the agent perform the heavy lifting by running simulations using different input features. For input features that perform well, we analyze the neural network and decipher the agent's replacement policy.

3) *Rigorously confirming the importance and sufficiency of heuristically-known features*: Though we considered non-obvious features like line address offset, set number, set accesses after miss, etc., our ML model picked features that heuristically have been known to be useful, such as reuse distance and hits since insertion. In addition to identifying features through the heat map analysis, we use hill-climbing analysis to

select a set of the most critical features, as described in Section 6.2.2. We also perform the analysis on benchmarks that show significant difference in LLC hit rate between Belady and LRU replacement policies. This rigorous approach proves the importance of selected features.

4) *New perspective on using features in a replacement policy*: By analyzing agent simulations, we identified a different approach for using some features in our replacement policy. For example, rather than segregating cache lines into clean and dirty, we use a cache line's access type to categorize it as prefetched or not. This allows us to predict whether the cache line will hit in the future, as described in Section 6.2.2.

5) *Automation of feature selection*: The entire process of feature selection, from agent simulation, neural network weight analysis, to hill climbing analysis was automated. Although in this paper we use heat map analysis for visual convenience, the weight comparison and feature count reduction were automated. Through this work, we show that ML is an effective tool for tackling challenging computer architecture design problems. An automated ML-based cache replacement policy can match or beat state-of-the-art hand-crafted designs.

For the cache replacement problem targeted in this work, we have the following insights.

1. Preuse distance can be used to estimate reuse distance of a cache line, which is essential for making good replacement decisions. This insight is drawn from the *line preuse* feature.

2. Cache lines loaded by prefetch accesses are reused within short time intervals. This insight is drawn from the *line last access type* feature. An efficient cache replacement policy can use this insight to evict non-reused prefetched cache lines.

3. A cache line that has been accessed multiple times is likely to be accessed again. This insight is drawn from the *line hits since insertion* feature.

4. Sometimes it is beneficial to evict the youngest cache line. This insight is drawn from the *line recency* feature.

6.3 Reinforcement Learned Replacement (RLR)

In this section, we propose a replacement policy (RLR) based on insights learned from the neural network. At a high level, RLR follows the following rules.

1. For a significant number of cache lines, the reuse distance can be approximated by preuse distance.
2. The type of previous access of a cache line can be used to predict its chance of receiving a hit.
3. Cache lines that have been accessed can be predicted to be accessed again in the future.
4. Recently-inserted cache lines are prioritized for eviction to allow older cache lines to be reused.

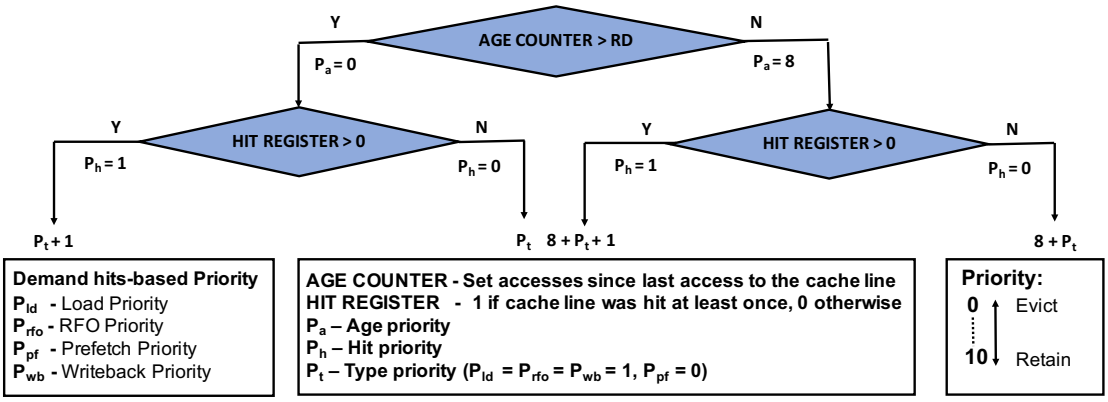


Figure 6.8: Flowchart for priority computation in RLR.

6.3.1 Replacement Algorithm

In RLR, reuse distance is predicted based on the preuse distance of the cache lines. Cache lines with age less than the predicted reuse distance are protected. In addition, cache lines are prioritized based on the type of their previous access and whether or not the cache lines have received hits. When a replacement decision is made, the cache lines

in the set are assigned priority levels. Priority levels are computed based on the cache line's age, previous access, and hits. On a cache miss, the cache line with the lowest priority will be evicted.

Age priority (P_{age}): Each cache line is augmented with an *Age Counter* that counts the line's age in set accesses (i.e., how many times the set has been accessed since the last access of the line). On a demand hit, the counter's value represents the preuse distance of the accessed cache line. Because we use preuse distance to approximate future reuse distance (Section 6.2.2), we predict that the cache line will be reused after a number of set accesses equal to the preuse distance. If the cache line is not accessed after the predicted reuse distance, it is considered for eviction. However, maintaining registers to store predicted reuse distance for each individual cache line is impractical. Instead, we accumulate the preuse distances of cache lines on demand hits and use the accumulated value to approximate future reuse distance (RD). Cache lines are protected from eviction until their respective ages reach RD. RD must be chosen carefully. On the one hand, if RD is too high, cache lines with small reuse distance might be retained in the cache longer than necessary. On the other hand, if RD is too low, cache lines with large reuse distance might be evicted prematurely before reuse. Also, reuse distance changes during application execution. Therefore, RD must be updated periodically to adapt to application phases. In our experiment, RD is updated for every 32 demand hits, by averaging the aggregated preuse distance and then multiplying by two (i.e., $RD = 2 \times Average\ Preuse\ Distance$). Recall that for most cache lines, the preuse distance and reuse distance are not exactly the same. Doubling the average preuse distance can be beneficial because it allows cache lines to stay in the cache longer.

Overall, age-based priority levels are assigned as follows.

- **Priority Level 1:** If Age Counter is smaller than RD, the cache line has not yet reached the reuse distance; this cache line might be reused in the future. Higher priority is assigned to retain the cache line for future reuse.
- **Priority Level 0:** If Age Counter is greater than RD, the cache line has already reached the reuse distance, but has not yet been reused since last access. Lower priority is assigned because the line might not be reused in the future.

Type priority (P_{type}): Each cache line is augmented with a *Type Register*, indicating

whether its previous access was a prefetch. Type-based priority levels are assigned as follows.

- **Priority Level 1:** If the last access type is not prefetch, then either the cache line was not inserted by a prefetch access, or it has been reused after insertion. We want to keep this cache line.
- **Priority Level 0:** If the last access type is prefetch, it has not been reused. When replacement is needed, we tend to evict non-reused prefetched cache lines.

Hit priority (P_{hit}): Each cache line is augmented with a *Hit Register* that is set when the cache line is hit. The hit-based priority levels are assigned as follows.

- **Priority Level 1:** If the Hit Register is 1, the cache line has been reused. We want to keep this cache line because the data can be accessed repeatedly in the same program.
- **Priority Level 0:** If the Hit Register is 0, the cache line has not been reused. When replacement is needed, we tend to evict non-reused cache lines.

The priority level for each cache line is computed as a weighted sum of the priorities described above, given by the following equation.

$$P_{line} = 8 \cdot P_{age} + P_{type} + P_{hit}$$

The weights are designed based on hill climbing analysis, described in insights. Agent performance was analyzed by enabling one feature and disabling the rest. Among all the features, preuse distance achieved the highest performance. Therefore, we assign it the highest weight. The features last access type and age contributed equally. Figure 6.8 shows the flowchart for priority computation in RLR. The age priority of a cache line is computed by comparing its Age Counter with RD. If the cache line's age is greater than RD, age priority (P_{age}) is set to 0, otherwise it is set to 8. The value 8 (for P_{age}) is chosen for two reasons. First, we give higher weights to cache lines whose age is less than RD, with the hope that they can be retained in the cache for a longer period and reused in the near future. Second, multiply by 8 can be implemented in hardware efficiently by left shifting three times. The hit priority P_{hit} of the cache line is computed

from the Hit Register. P_{hit} is 1 if the Hit Register is set; otherwise it is 0. The type priority P_{type} of the cache line is computed from the Type Register. P_{type} is 0 if the Type Register indicates a prefetch access; otherwise it is 1.

To select a replacement candidate, the cache management policy selects the way with the lowest priority level. It is possible that multiple ways have the same priority level. To break ties, we use the recency information.

Recency: Each cache line has a $log(Set\ Associativity)$-bit value indicating the relative order of access among the lines in a set. When multiple lines have the same priority, the most recently accessed cache line (high recency value) is evicted. The most recently accessed cache line takes the longest time to reach the RD value. Evicting it allows the other cache lines to reach the RD value. If cache bypass is supported, the cache management policy bypasses a request if no cache line has reached an age greater than the RD value. In RLR, recency plays an important role in victim selection. However, tracking recency accurately for every cache line can be costly. In Section 6.3.3, we describe an optimization technique to represent recency using fewer bits.

6.3.2 Hardware Implementation

Each cache line is equipped with an Age Counter, a Hit Regsiter, and a Type Register. The Age Counter is an n-bit saturating counter, tracking 2^n number of set accesses. When a demand hit occurs, the cache line's Age Counter is sent to the Accumulator. After the number of demand hits reaches a threshold (32 in this case), we average the accumulated value and then double the value. The averaging circuit can be as simple as a right shifting logic, as long as the demand hit threshold is a power of 2. For example, to average over $2^5 = 32$ cache hits, the accumulator value is right shift by 5. Also, the averaged preuse distance can be doubled by left shifting 1 bit. We combine the averaging and doubling circuit by right shifting the accumulated value by $(5 - 1) = 4$ bits.

The hardware implementation for computing RD is shown in Figure 6.9. The Hit Register of a cache line is set when it receives a hit. A Type Register is used to indicate if the cache line's previous access type was prefetch or not. To estimate the hardware cost of RLR, we synthesize the design in 32 nm technology using Synopsys Design Compiler [27]. The area, power, and latency for RLR are $84\mu m^2$, $4.6\mu W$, and

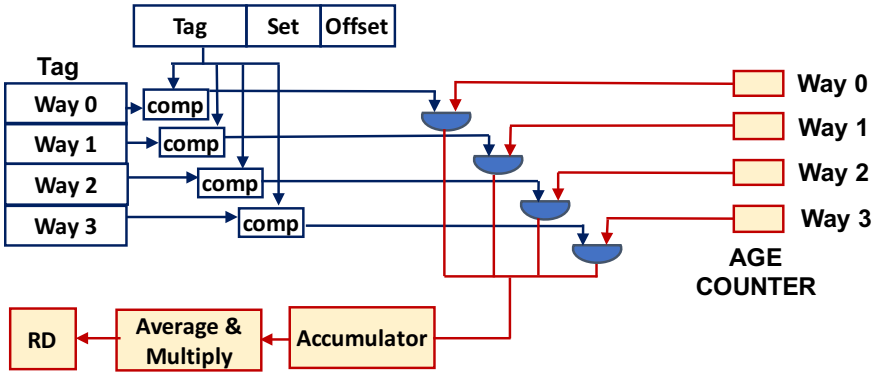


Figure 6.9: Hardware implementation for computing RD. On a demand hit, the cache line's age value is sent to the RD computation circuit.

0.68ns, respectively. The cache line priority computation in RLR can be done in parallel to the tag comparison (to determine hit/miss) and does not contribute to the critical path latency. RD computation in RLR is done when LLC is idle or fetching data from memory, so that it is not part of the critical path. The area overhead is negligible compared to total processor area. For example, the area of Intel Sandy Bridge processor with similar configuration is 216 mm^2 [82].

6.3.3 Optimizations

To determine the Age Counter's optimal size, we ran simulations by varying the number of Age Counter bits from 2 to 8 bits per cache line. To achieve good performance while keeping the overhead low, we chose 5 bits to represent Age Counter. We verified that 5 bits are sufficient to cover the average preuse distance in agent simulations for most benchmarks. In addition, we used a 1-bit Hit Register, a 1-bit Type Register, and $log(Set\ Associativity)$ bits for recency of a cache line. In a 16-way set associative cache, this amounts to 11 bits of overhead per cache line. To further reduce overhead, we devised two optimizations.

Age Counter Optimization: There are two ways to minimize the overhead of the Age Counter – counting fewer events and approximating counter value. To count fewer events, we use Age Counter to count the number of set misses instead of set accesses. After a hit, cache lines in a set remain unchanged. By counting set misses, Age Counter

represents the relative age of a cache line (since its last access) rather than the absolute age. To approximate counter value, we increment Age Counter for every 8 set misses. This allows us to reduce the overhead per line by 3 bits. We use a 3-bit counter per set to count every set miss. After 8 set misses, the counter rolls over and the line counters are incremented.

Recency Approximation: We can use the age of a cache line to determine its recency. The most recently accessed cache line is either hit or inserted in the cache. In both cases, the age of the line is zero. For a hit, the age counter value is sent to the accumulator for computing RD, then reset. On a miss, a new line replaces a victim, and the corresponding age counter is reset. Therefore, the most recently accessed line can be identified by age counter value zero. In a 16-way set associative cache, using age counter to determine recency reduces the overhead by 4 bits per cache line. In RLR, recency is used to break ties when multiple cache lines have the same priority level. When multiple cache lines have the same age counter value and the lowest priority level, we chose to evict the cache line with the lowest way index.

In summary, after optimization, each cache line has a 2-bit Age Counter, a 1-bit Hit Register, and a 1-bit Type Register, totaling 4 bits of overhead per cache line. In addition, we use a 3-bit counter per set. For a 2MB 16-way LLC with 64B cache line, the total storage overhead of RLR is 16.75KB.

6.3.4 Multicore Extension

In a multicore system that executes different benchmarks on separate cores, cache lines in the LLC can be segregated based on the ids of the cores that issue the accessed requests. Although each core (benchmark) has a different reuse characteristic, it is challenging to predict their behavior when accesses from all cores are mixed. This is because two consecutive accesses from the same core can be interleaved by multiple accesses from other cores. However, we observe that the access frequency of a core and its average reuse distance have an inverse correlation. That is, a core that has the most number of LLC accesses within a time interval also has the smallest average reuse distance. This is because a cache line from a core having high access frequency tends to be reused earlier than a cache line from a core with low access frequency. This information can be used in the replacement policy to select a victim with large reuse

Table 6.4: Parameters for the evaluated system

Core	6-stage pipeline, 3-issue O3, 256-entry ROB
L1 I-Cache	32 KB, 8-way, 4-cycle latency, LRU
L1 D-Cache	32 KB, 8-way, 4-cycle latency, LRU
L2 Cache	256 KB, 8-way, 12-cycle latency, LRU
LLC (per core)	2 MB, 16-way,26-cycle latency
Prefetcher	next-line (L1), IP-stride (L2), None (LLC)

distance by selecting a cache line from a core with low access frequency.

In RLR, we assign priorities for each core. When a replacement decision is made, each cache line in the set is assigned a priority based on its age, hit, type and core. The cache line with the lowest priority is chosen for eviction. Based on our experiments, assigning core priorities based on demand hit frequency instead of access frequency yields better performance. For this, we maintain demand hit counters for each core at the LLC. On every demand hit (Load or RFO hit), the LLC demand hit counter corresponding to the core of the cache line is incremented. Based on the number of demand hits, each core is assigned a **Priority Level (0, 1, 2, or 3)**. A core with more demand hits is assigned a higher priority level. The core priorities (P_{core}) are updated for every 2000 LLC accesses. In terms of overhead, the demand hit counters contribute 12 bits per core to the overall storage overhead. The priority level for each cache line (P_{line}) in a set is computed by the following formula.

$$P_{line} = 8 \cdot P_{age} + P_{type} + P_{hit} + P_{core}$$

6.4 Evaluation

In this section, we evaluate the effectiveness of RLR against other contemporary cache replacement policies. We implement RLR in ChampSim simulator from the 2^{nd} Cache Replacement Championship (CRC2). We estimate the performance on 1-core and 4-core configurations with a 6-stage pipeline and a 256-entry reorder buffer. The memory system has a 3-level cache hierarchy with private L1, L2 and a shared LLC. The complete configuration is shown in Table 6.4. We use SPEC CPU® 2006 [83] and CloudSuite [84] benchmarks for evaluation. To train the RL agent, we only used the first 100M instructions of eight SPEC CPU benchmarks. In evaluation, however, we also show results for

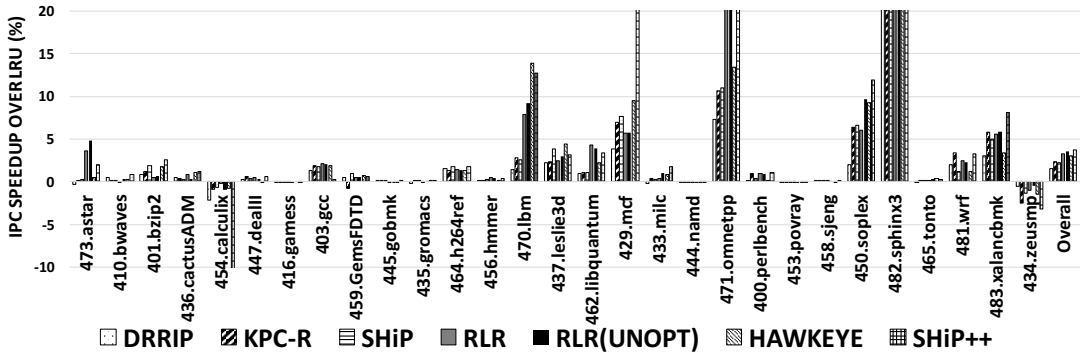


Figure 6.10: Performance comparison for different LLC replacement policies (SPEC2006).

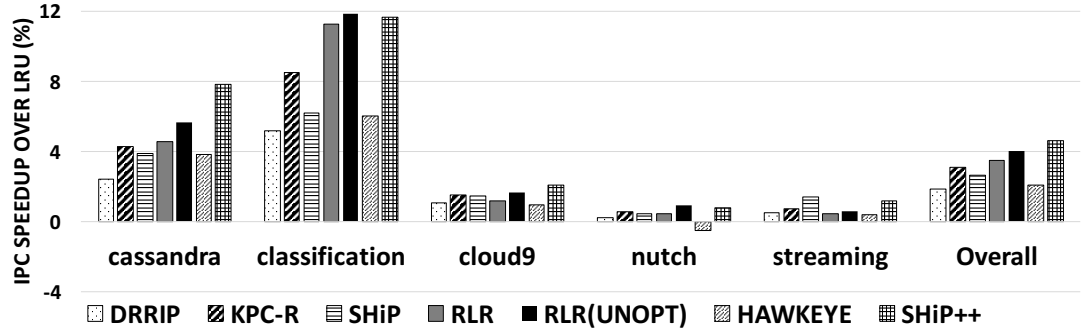


Figure 6.11: Performance comparison for different policies (Cloudsuite).

26 new benchmarks that have not been used in training.

For SPEC CPU 2006 evaluation, we use all 1 billion instruction traces from SimPoint [85] provided by CRC2. For Cloudsuite benchmarks, we use all traces files provided by CRC2. We warm the cache for 200 million instructions and evaluate the performance for the next one billion instructions. In 4-core simulations, we evaluate performance when four different benchmarks are run simultaneously on separate cores. We generate 100 random sets of four benchmarks from the 29 applications in SPEC CPU 2006. We use the same trace files as in single-core simulations. However, we run the simulation until each benchmark completes one billion instructions. If any benchmark reaches the end of its trace, the corresponding core continues simulating from the beginning of the trace file. In 4-core simulations for Cloudsuite benchmarks, we run each trace in its respective core. For the results of single-core simulations, we use IPC speedup over LRU. The IPC speedup of each benchmark i is measured as $\frac{IPC_i}{IPC_{i,LRU}}$. If a benchmark has more than one trace file, we present the geometric mean of all IPC speedup numbers. For multicore results, the overall performance of each workload mix is measured as the geometric mean of IPC speedups of all benchmarks in the mix $(\prod_{i=1}^{4} \frac{IPC_i}{IPC_{i,LRU}})^{\frac{1}{4}}$.

We compare the performance of RLR against KPC-R, DRRIP, and SHiP, as well as policies from CRC2, including SHiP++ and Hawkeye. We obtained the source code for these policies from the CRC2 website. We also compare against some prior works such as EVA [86] and PDP [87].[2] Compared to LRU, we observed IPC degradation in both EVA and PDP. For SPEC CPU 2006, EVA degrades single-core system performance by 0.11%, and PDP degrades performance by 3.72%, on average. The original works on EVA [86] and PDP [87] show performance improvements with respect to LRU and DRRIP. Considering that the margin of difference in performance between any two LLC replacement policies is small, the selection of instruction traces used for evaluation can have significant impact on overall results. Using a rigorous evaluation methodology is important. Performance discrepancies for EVA and PDP may be attributed to use of single instruction traces, not based on SimPoints [85], that do not fully characterize an entire application. We observed that overall ranking of policies can change between

[2]We procured EVA source code from http://people.csail.mit.edu/sanchez. We implemented PDP as described in [87].

individual SimPoints. As such, we compute results from all SimPoints to ensure accurate representation of benchmark behavior.

In addition to results for overhead-optimized policies, we also present the results of RLR without overhead reduction optimizations (Section 6.3.3), denoted by RLR(unopt). In the unoptimized policy, we use a 2-bit Hit Counter, as opposed to a 1-bit Hit Register. Each cache line has a 5-bit Age Counter, a 2-bit Hit Counter, and a 1-bit Type Register, amounting to 10-bit overhead per cache line. For a 2MB 16-way LLC, the total storage overhead is 40KB for the unoptimized policy.

Figures 6.10 and 6.11 show performance comparisons for SPEC CPU 2006 and Cloudsuite benchmarks, respectively. RLR outperforms KPC-R and DRRIP for all benchmarks. For our evaluations, we used the IP-stride prefetching policy at L2. Since the performance of KPC-R is improved by information from KPC-P prefetching, we also compared KPC-R and RLR with KPC-P as the L2 prefetching policy. In such a system, KPC-R and RLR improve performance by 3.9% and 5.5%, respectively, for SPEC CPU 2006. For Cloudsuite, KPC-R and RLR improve performance by 2.46% and 3.5%, respectively. RLR performs better than KPC-R by evicting non-reused prefetched cache lines in LLC sooner. KPC-P avoids cache pollution in two ways. First, low-confidence prefetches are not inserted in L2. Second, when a prefetch access hits in LLC, the corresponding cache line is promoted in the replacement stack only when the prefetch confidence is higher than a threshold. While the first method prevents cache pollution in L2, the second method does not evict non-reused prefetched cache lines in LLC sooner than cache lines from other access types. RLR also outperforms SHiP for most benchmarks. For example, in 470.lbm, prefetching does not improve IPC significantly. RLR evicts prefetched cache lines sooner than other cache lines, resulting in better performance compared to SHiP. In memory-intensive benchmarks such as 429.mcf, SHiP maximizes hit rate by ranking PCs in the order of hit contribution and retaining cache lines fetched by highly ranked PCs. For a few benchmarks, like 437.leslie3d and streaming (Cloudsuite), RLR has a larger number of LLC demand hits (Load and RFO) compared to SHiP. However, the trend does not reflect in IPC. This is because SHiP is more likely to retain cache lines from PCs contributing to IPC improvement, while RLR retains cache lines contributing to demand hits.

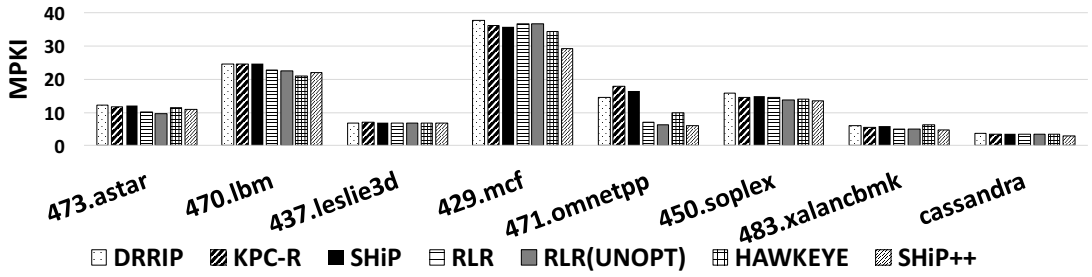


Figure 6.12: Demand MPKI comparison for different policies.

Table 6.5: Overall speedup for different replacement policies.

Policy	1-core (2MB LLC)		4-core (8MB LLC)	
	SPEC 2006	Cloud-Suite	SPEC 2006	Cloud-Suite
DRRIP	1.50 %	1.80 %	2.63 %	1.07 %
KPC-R	2.30 %	3.07 %	5.50 %	3.80 %
RLR	3.25 %	3.48 %	4.86 %	2.39 %
RLR(unopt)	3.60 %	4.02 %	5.87 %	2.5 %
SHiP	2.24 %	2.64 %	6.33 %	3.09 %
Hawkeye	3.03 %	2.09 %	7.69 %	2.45 %
SHiP++	3.76 %	4.60 %	7.37 %	3.89 %

Table 6.5 summarizes performance improvement over LRU for evaluated replacement policies for single-core and multicore workloads. For single-core and multicore evaluation, overall performance improvement is computed as the geometric mean of IPC speedup for all evaluated workloads. In single-core evaluations, RLR outperforms DRRIP, KPC-R, SHiP (PC-based), and Hawkeye (PC-based) by 1.74%, 0.95%, 1.01%, and 0.22%, respectively, for SPEC CPU 2006. In Cloudsuite, RLR outperforms DRRIP, KPC-R, SHiP (PC-based), and Hawkeye (PC-based) by 1.65%, 0.41%, 0.84%, and 1.39%, respectively. Figure 6.12 shows Misses Per Kilo-Instructions (MPKI) for benchmarks with MPKI greater than 3. Compared to DRRIP, RLR achieves a maximum of 52% reduction in 471.omnetpp and a minimum of 2.5% in 429.mcf.

We simulated policy variants by eliminating hit and type priorities to evaluate their contributions to RLR's performance. In SPEC CPU 2006, IPC speedup over LRU reduces by 12% when the hit register is disabled. This shows that protecting cache lines that received at least one hit over the cache lines that were never hit has a significant

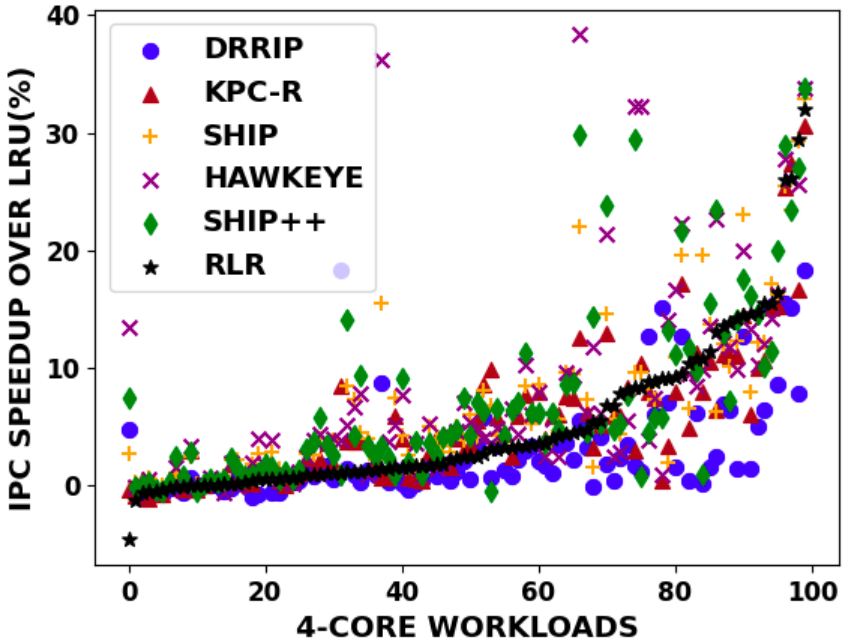


Figure 6.13: Performance comparison of different policies in the 4-core setup.

impact on performance. When the type register is disabled, speedup reduces by 30%, demonstrating that significant performance gains are achieved by protecting cache lines from one access type over another.

In multicore evaluation (Figure 6.13), RLR outperforms DRRIP by 2.3% and 1.32% in SPEC2006 and Cloudsuite, respectively. Contrary to the single-core results, KPC-R outperforms RLR by 0.64% and 1.41% in SPEC2006 and Cloudsuite, respectively. This is because interference from other core accesses delays the reuse of prefetched cache lines. However, RLR allows prefetched cache lines to be reused within short time intervals. RLR performance can be improved by considering each core's access frequency for the eviction of prefetched cache lines. SHiP also performs better than RLR by 1.32% and 0.7% in SPEC2006 and Cloudsuite, respectively. Analyzing the workloads in which SHiP outperforms RLR reveals that in some workloads, a small percentage of PCs account for nearly all demand hits. This application characteristic allows any PC-based policy to protect cache lines frequently accessed by those small percentage of PCs while evicting cache lines brought in by other PCs. Given the large number of LLC accesses in a multicore system, the information brought by PCs from

different cores is useful. Though we lose some information by avoiding PC usage, RLR captures the benchmarks characteristics through features that can be computed readily at the LLC and achieves a performance similar to the PC-based policies. Using KPC-P instead of IP-stride prefetching policy, RLR outperforms KPC-R by 0.5%, indicating that without using PC in the memory system, RLR performs better than other non-PC based replacement policies. Also, with RLR, we avoid the complexity of designing and verifying a multicore system that incorporates hardware infrastructure for accessing PC at LLC. We have the luxury of building a standalone cache design that can be integrated with already designed and verified single/multi-core systems.

6.5 Related Work

There has been considerable work on designing efficient cache replacement algorithms [17–22]. In this section, we discuss some of the related prior work.

Glider: Glider [68] uses a Support Vector Machine (SVM)-based hardware predictor for cache replacement. Initially, an offline attention-based long short-term memory (LSTM) model is used to improve prediction accuracy. Then, the authors interpret the offline model to gain insights and hand craft a feature that represents a program's control-flow history. Then, a simple linear learning model is used to match the LSTM's prediction accuracy. The hardware implementation of the policy requires a Program Counter History Register and an Integer SVM table.

Hawkeye: Hawkeye [67] uses a PC-based predictor to determine whether a cache line is cache-friendly or cache-averse. On a miss, the policy first chooses to evict cache-averse lines. If no cache-averse cache lines are found in the set, the oldest accessed cache line is chosen for eviction.

SHiP: Signature Based Hit Prediction (SHiP) [65] replacement policy predicts re-reference characteristics of cache lines from a PC-based signature. SHiP has a Signature History Counter Table (SHCT) that maintains a counter (SHCTR) for each PC signature. Cache lines inserted by PCs with non-zero SHCTR are assigned a Re-Reference Prediction Value (RRPV) of 2, while other cache lines are inserted with a RRPV of 3.

SHiP++: The SHiP++ replacement policy [66] enhances SHiP by inserting cache lines accessed by PCs with maximum SHCTR value with a RRPV of 0, training the SHCT

table only on the first re-reference of a cache line, inserting cache lines from writeback accesses with a RRPV of 3, assigning a separate PC signature for prefetch accesses, and making prefetch-aware RRPV updates on a cache line re-reference.

Counter based replacement policy: Kharbutli *et al.* [88] propose a counter-based approach, where each cache line is equipped with counters to track events such as the number of accesses to the set between two consecutive cache line accesses or the number of cache line accesses. When the counter reaches a threshold, the line is eligible for replacement. The policy also uses a PC-based prediction table to retain counters for evicted cache lines.

All of the above policies correlate the reuse behavior of cache lines to the PCs. However, accessing PC at the LLC adds significant hardware overhead to the architecture. The width of the data path must be increased to propagate PC through all levels of the cache hierarchy. In addition, the miss status holding registers (MSHR) also need to track the PC information. These modifications exacerbate energy and communication overheads. Furthermore, pipeline design must be modified to propagate PC through all stages, as well as adding extra storage at Load/Store Queue before accessing the memory system. While many of the prior works lack the justification of the hardware and processor design/verification overhead associated with incorporating PC in cache replacement, it is questionable whether the benefits will outweigh the overhead and complexity. As a result, we avoid using PC in our replacement policy.

KPC: Kill the Program Counter [69] proposes an integrated data prefetching and replacement policy that avoids using the PC. While our work aims to design an effective replacement policy given an existing prefetcher, KPC designs a custom prefetcher (i.e., KPC-P) to complement their replacement policy. KPC-P uses prediction confidence to estimate reuse distance for prefetched cache lines and determine the cache level in which to insert them. The KPC cache replacement policy (i.e., KPC-R) uses two global counters to adapt to the dynamic program phase and decide whether to insert a cache line in LRU (RRPV=3) or near LRU (RRPV=2) positions in the replacement stack. KPC avoids L2 cache pollution by not inserting prefetched lines with low prediction confidence in L2; however, all prefetched lines are inserted in LLC. As described in Section 6.2.2, we avoid cache pollution from prefetched cache lines in LLC by evicting non-reused prefetched cache lines sooner than cache lines from other access types.

The following replacement policies use past accesses of a cache line to predict its future access behavior. Each policy predicts a cache line's reuse characteristic using a certain metric. On a miss, all cache lines in the accessed set are compared using the metric, and the cache line with the farthest reuse characteristic is chosen for eviction.

PDP: Protecting Distance based Policy (PDP) [87] protects all lines in LLC until the number of accesses to the set (after the line insertion or access) reaches a threshold, known as Protecting Distance (PD). On a miss, an unprotected cache line is evicted. If no unprotected cache lines are found, either the cache line with the minimum number of set accesses is evicted or the access is bypassed. A dedicated special-purpose processor executes a search algorithm periodically to compute the optimal threshold. Hit rates are estimated for threshold values less than 256, and the threshold value with the best hit rate is chosen. In our work, we derive a much simpler method to predict reuse distance (like PD in [87]) based on insights gained from the ML model.

EVA: The Economic Value Added (EVA) metric [86] characterizes the difference between expected and average hits. A cache line's EVA depends on its age, and the cache line with the lowest EVA is evicted. However, the policy does not account for non-demand accesses, such as prefetch accesses. These additional accesses may skew the correlation between a cache line's age and its EVA.

RWP: Read-Write Partitioning (RWP) [22] is a replacement policy that dynamically partitions the cache into clean and dirty partitions to reduce the number of read misses. On a miss, a victim is selected from one of the partitions, based on predicted partition size and the actual partition size in the corresponding set.

Inter-reference Gap Distribution Replacement [89] uses time difference between successive references of a cache line to attach a weight to it. On a miss, the cache line with the smallest weight is evicted. Das *et al.* [90] propose using a cache line's age (since last access) to estimate its hit probability under an optimal replacement policy. On a miss, the line with the lowest hit probability is evicted. Keramidas *et al.* [91] combine the usage of reuse distance and PC. The policy uses a sampler to compute reuse distances for selected cache lines. The computed reuse distances and the PCs of load/store instructions that accessed the selected cache lines are used to predict reuse distances for other cache lines. On a miss, the cache line with largest predicted reuse distance is evicted.

6.6 Summary

Machine learning is useful in architecture design exploration. However, human expertise is still essential in deciphering the ML model, making design trade-offs, and finding practical solutions. In this work, with the goal of designing a cost-effective cache replacement policy, we used reinforcement learning to guide and expedite our design. We trained an RL agent using features that are relatively easy to obtain at the LLC. Considering the complexity in propagating PC information to the LLC, we intentionally excluded PC from the feature set. After training the agent neural network, we identified important features from a large feature set by analyzing neural network weights. Based on insights drawn from the neural network, we successfully derived a new replacement policy. We then optimized the proposed policy to further reduce hardware overhead. Overall, the proposed replacement policy outperforms DRRIP (non-PC-based policy) and achieves comparable performance to existing PC-based replacement policies.

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.

We further exploited the efficiency of ML in solving another complex computer architecture problem. Using the ability of ML to expedite tedious processes and augment human intelligence, we automated the process of generating architectural insights and developing a high-performance cache replacement policy. In our work, we used reinforcement learning as an offline tool and allowed the RL agent to learn a near-optimal replacement policy using memory-intensive benchmarks. We then analyzed the RL agent to gain new insights and developed a cost-effective replacement policy that has a similar performance to the RL-learned policy. The proposed replacement policy beats the performance of current state-of-art policies. While our methodology used a cache replacement policy as an example, a similar approach can be used to solve other architecture problems in which heuristic-based solutions are limited by designer time and creativity. A few examples where a similar approach can prove useful are prefetching, warp- or block-level scheduling in GPUs, instruction issuing, and many more.

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 representation 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] Samira M Khan, Yingying Tian, and Daniel A Jiménez. Dead block replacement and bypass with a sampling predictor. In *MICRO*, 2010.

[18] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO*, 2008.

[19] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[20] Carole-Jean Wu and Margaret Martonosi. Adaptive timekeeping replacement: Fine-grained capacity management for shared cmp caches. *TACO*, 8(1), 2011.

[21] Viacheslav V Fedorov, Sheng Qiu, AL Narasimha Reddy, and Paul V Gratz. Ari: Adaptive llc-memory traffic management. *TACO*, 10(4), 2013.

[22] Samira Khan, Alaa R Alameldeen, Chris Wilkerson, Onur Mutluy, and Daniel A Jimenezz. Improving cache performance using read-write partitioning. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 452–463. IEEE, 2014.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] Cadence. *Encounter User Guide.*

[29] 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.

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

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

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

[33] 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.

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

[35] 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.

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

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

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

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

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

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

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

[62] Nan Wu and Yuan Xie. A survey of machine learning for computer architecture and systems. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

[63] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3), 2010.

[64] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *MICRO*, 2017.

[65] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *MICRO*, 2011.

[66] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moin Qureshi. Ship++: Enhancing signature-based hit predictor for improved cache performance. In *CRC*, 2017.

[67] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ISCA*, 2016.

[68] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *MICRO*, 2019.

[69] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. 2017.

[70] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[71] Self optimizing memory controllers: A reinforcementlearning approach. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[72] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *MICRO*, 2016.

[73] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *MEMSYS*, 2017.

[74] Hao Zheng and Ahmed Louri. An energy-efficient network-on-chip design using reinforcement learning. In *DAC*, 2019.

[75] Quintin Fettes, Mark Clark, Razvan Bunescu, Avinash Karanth, and Ahmed Louri. Dynamic voltage and frequency scaling in nocs with supervised and reinforcement learning techniques. *IEEE Transactions on Computers*, 68(3), 2018.

[76] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, Mark Oskin, Natalie Enright Jerger, and Gabriel H Loh. Experiences with ml-driven design: A noc case study. In *HPCA*, 2020.

[77] Jieming Yin, Yasuko Eckert, Shuai Che, Mark Oskin, and Gabriel H. Loh. Toward more efficient noc arbitration: A deep reinforcement learning approach. *AIDArch 2018*.

[78] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.

[79] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, 2005.

[80] The 2nd cache replacement championship. `https://crc2.ece.tamu.edu/`, 2017.

[81] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

[82] Henry. Intel 32nm-22nm comparison. `http://blog.stuffedcow.net/2012/10/intel32nm-22nm-core-i5-comparison`.

[83] SPEC CPU 2006. `https://www.spec.org/cpu2006/`.

[84] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *ACM SIGPLAN Notices*, 47(4), 2012.

[85] SimPoint. `http://www.cs.ucsd.edu/users/calder/simpoint/`.

[86] Nathan Beckmann and Daniel Sanchez. Maximizing cache performance under uncertainty. In *HPCA*, 2017.

[87] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *MICRO*, 2012.

[88] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4), 2008.

[89] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Supercomputing*, 2004.

[90] Subhasis Das, Tor M Aamodt, and William J Dally. Reuse distance-based probabilistic cache replacement. *TACO*, 12(4), 2015.

[91] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, 2007.