Leveraging machine learning for managing prefetchers and designing secure standard cells

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

LEVERAGING MACHINE LEARNING FOR MANAGING PREFETCHERS AND DESIGNING SECURE STANDARD CELLS

by

FURKAN ERIS

B.S., Bogazici University, 2016
M.S., Boston University, 2020

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2022

Approved by

First Reader

Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

Second Reader

Paul Keltcher, PhD
Principal Member of Technical Staff
Advanced Micro Devices, Inc.

Third Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Fourth Reader

M. Selim Ünlü, PhD
Professor of Electrical and Computer Engineering
Professor of Materials Science and Engineering
Professor of Biomedical Engineering

Calvin: Wow it really snowed last night! Isn't it wonderful?
Hobbes: Everything familiar has disappeared! The world looks brand-new!
Calvin: A new year… A fresh clean start!
Hobbes: It's like having a big white sheet of paper to draw on!
Calvin: A day full of possibilities!
Calvin: It's a magical world Hobbes ol' buddy…
Calvin: … Lets go exploring!

– Bill Watterson (1995)

Acknowledgments

First, I would like to express my appreciation and gratitude to my advisor, Prof. Ajay Joshi, for his continuous support and guidance throughout my Ph.D. His guidance made me a more resilient researcher. I want to thank Dr. Paul Keltcher for his patience and mentorship at AMD. Additionally, I like to extend my gratitude to the rest of my thesis committee members, Prof. Martin Herbordt and Prof. Selim Unlu, for their precious time, generous support, and insightful feedback.

I am very grateful to all my collaborators and co-authors, especially Sadullah Canakci, Marcia Sahaya Louis, Aditya Narayan, Cansu Demirkiran, and Dr. Yenai Ma. I will fondly remember my chocolate sessions at AMD with Sadullah and his friendship throughout the years. I would like to thank all of my friends in BU ICSG research group and BU PEAC Lab. Next, I would like to thank decade-old friendships that are as strong as ever even after six years across seven time-zones with Alican Akman, Goktug Bacanli, and Mustafa Emin Oktay. Emin is much closer now, Alican and Goktug are next! I truly appreciate the friendships I have gained with everyone at Boston University, but I have to give a special note for Burak Aksar who has been a partner in crime for me through these past four years. I want to thank my parents and little brother Selman for their unconditional love and support, for being my source of inspiration and encouragement, and for constantly providing the mental aid that I required. It is amazing how lucky I have been for having been born into this family. Finally, the love of my life Kubra. Without her, none of this would have been possible. I would have given up in the first couple of years. She deserves this as much as I do. There are not enough words to express the love I feel for her. Every day is a new adventure.

LEVERAGING MACHINE LEARNING FOR MANAGING PREFETCHERS AND DESIGNING SECURE STANDARD CELLS

FURKAN ERIS

Boston University, College of Engineering, 2022

Major Professor: Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

Machine Learning (ML) has gained prominence in recent years and is currently being used in a wide range of applications. Researchers have achieved impressive results at or beyond human levels in image processing, voice recognition, and natural language processing applications. Over the past several years, there has been a lot of work in the area of designing efficient hardware for ML applications. Realizing the power of ML over the years, lately, researchers are exploring the use of ML for designing computing systems. In this thesis, we propose two ML-based design and management approaches - in the first approach, we propose to use ML algorithms to improve hardware prefetching in processors. In the second approach, we leverage Reinforcement Learning (RL)-based algorithms to automatically insert nanoantennas into standard cell libraries to secure them against Hardware Trojans (HTs).

In the first approach, we propose using ML to manage prefetchers and in turn improve processor performance. Classically, prefetcher improvements have been focused on either adding new prefetchers to the existing hybrid prefetching system (a system made out of one or more prefetchers) or increasing the complexity of the existing prefetchers. Both approaches increase the number of prefetcher system configurations (PSCs). Here, a PSC is a given setting for each prefetcher such as whether it is ON or OFF or in the case of more complex prefetchers settings such as the aggressiveness level of the prefetcher. While the choice of PSC of the hybrid prefetching system can be statically optimized for the average case, there are still opportunities to improve the performance at runtime. To this end, we propose a prefetcher manager called Puppeteer to enable dynamic configuration of existing prefetchers. Puppeteer uses a suite of decision trees to adapt PSCs at runtime. We extensively test Puppeteer using a cycle-accurate simulator across 232 traces. We show up to 46.0% instructions-per-cycle (IPC) improvement over no prefetching in 1C, 25.8% in 4C, and 11.9% over 8C. We design Puppeteer using pruning methods to reduce the hardware overhead and ensure feasibility by reducing the overhead to only a few KB for storage.

In the second approach, we propose SecRLCAD, an RL-based Computer-Aided-Design (CAD) flow to secure standard cell libraries. The chip supply chain has become globalized. This globalization has raised security concerns since each step in the chip design, fabrication and testing is now prone to attacks. Prior work has shown that a HT in the form of a single capacitor with a couple of gates can be inserted during the fabrication step and then later be utilized to gain privileged access to a processor. To combat this inserted HT, nanoantennas can be inserted strategically in standard cells to create an optical signature of the chip. However, inserting these nanoantennas is difficult and time-consuming. To aid human designers in speeding up the design of secure standard cells, we design an RL-based flow to insert nanoantennas into each standard cell in a library. We evaluate our flow using Nangate FreePDK 45nm. We can secure and generate a clean library with an average area increase of 56%.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
AMD	Advance Micro Devices
ASIC	Application-specific integrated circuit
CAD	Computer Aided Design
CMOS	Complementary Metal Oxide Semiconductor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DRC	Design Rule Constraint
DVFS	Dynamic Voltage Frequency Scaling
DQN	Deep Q-Network
FDTD	Finite Difference Time Domain
FSD	Full Self Driving
GCN	Graph Convolutional Network
GDS	Graphic Design System
GF	GlobalFoundries
GND	Ground
GNN	Graph Neural Network
GPU	Graphics Processing Unit
HPC	Hardware Performance Counter
HPE	Hewlett Packard Enterprise
HT	Hardware Trojan
IC	Integrated Circuit
IP	Intellectual Property
IR	Infrared
IPC	Instructions Per Cycle
LEF	Library Exchange Format
LIB	Liberty File
ML	Machine Learning
NN	Neural Network
OS	Operating System
OoO	Out of Order
PIM	Processing in Memory
PEX	Parasitic Extraction
PnR	Place and Route
PSC	Prefetcher System Configuration

ReLU	Rectified Linear Unit
RF	Random Forest
RL	Reinforcement Learning
SEM	Scanning Electron Microscope
SoC	System on Chip
SP	Spice File

Chapter 1

Introduction

1.1 Motivation

The demand for computing is continuously increasing with no signs of slowing down. High-performance computing, cloud computing, and end user computing demand markets are all growing year-after-year. High-performance demand has pushed companies such as Advanced Micro Devices (AMD), Hewlett Packard Enterprise (HPE), and Cray Computing to develop Exascale computers (HPE, Cray, AMD, 2021). Reportedly, cloud computing market has increased from a 313 billion dollar to 483 billion in just two years, from 2020 to 2022 (Forbes, 2021). The end user computing market is expected to keep growing each year by 12.18% (Research Markets, 2022). With new application domains in computing such as autonomous vehicles, virtual/augmented reality, and bitcoin, to name a few, the computing industry has more challenges and questions than ever before. For example, on the energy consumption side, buying something as simple as a latte using bitcoin consumes up to 176 dollars in energy per transaction (Fortune, 2021). Adding all these transactions up, the global energy consumption from only bitcoin accumulates to 110 Terawatts per hour (HBR, 2021). More power efficient methods must be developed by computer scientists, researchers, and companies alike in order to tackle these issues. Security is becoming more prevalent as attacks have exposed hardware faults such as the Spectre (Kocher et al., 2019) and Meltdown (Lipp et al., 2018) attacks in 2019, which exploited critical architectural vulnerabilities in modern processors. These vulnerabilities were used in both Intel and AMD processors to leak confidential information such as passwords and personal data.

The computing research community must find reliable and effective methods to secure both software and hardware against these types of attacks before they cause irreversible harm. Finally, there are performance-critical safety issues, such as in the automotive industry where autonomous driving applications have to make decisions in fractions of nanoseconds with near-perfect accuracy. Companies like Tesla have designed chips such as the FSD that have significant safety components like redundant cores to cross-check and validate self-driving decisions (Drive Tesla Canada, 2021). Essentially as new domains have opened up, questions related to performance, energy consumption, safety, and security have become even more challenging. All of these challenges have been further exacerbated with the impending demise of technology scaling (Moore, 2019). There is no longer a free lunch with the periodic performance improvement stemming from technology scaling.

Research has instead shifted to using "More-than-Moore" approaches to sustain the historical trend of performance improvement and keep up with the wide variety of new computing domains. Exotic technology solutions such as 2.5D SoCs (Coskun et al., 2018; Coskun et al., 2020; Ma et al., 2021; Eris et al., 2018), 3D die stacking (Black et al., 2006; Naffziger et al., 2021), photonics, and optical enabled SoCs (Narayan et al., 2021; Guo et al., 2019; Demirkiran et al., 2021), quantum computing (Jurcevic et al., 2021), and many more. The architectural-level approaches include the design of domain-specific architectures for various domains (Dally et al., 2020), including but not limited to low-power sensing applications (Dong et al., 1997; Hempstead et al., 2005; Tekeste et al., 2018; Silva et al., 2018), graphics applications (Jouppi et al., 2017; Aamodt et al., 2018; Tran and Cambria, 2018), automotive applications (Talpes et al., 2020; Xu et al., 2021b; Du et al., 2019), etc.

With increasingly complicated and diversified computing systems, it is exceedingly difficult for human researchers to develop heuristic methodologies that work for all potential cases. In recent years, we have begun to observe research in utilizing ML to help researchers design and manage the hardware. The interest in this trend is increasing in both academia and in industry utilizing ML for both design and management of hardware (AnalyticsVidhya, 2021; Pan, 2021; Mirhoseini et al., 2021). Now by leveraging the power of ML, we can come up with novel designs (Yu et al., 2018; Mirhoseini et al., 2021; Yu and Zhou, 2020; Lin et al., 2020; Ustun et al., 2020; Ren and Fojtik, 2021; Ren et al., 2021) and control policies for hardware (Dean et al., 2018; Dean, 2020; Jiménez and Teran, 2017; Bhatia et al., 2019; Liao et al., 2009; Ipek et al., 2008; Tarsa et al., 2019; Ravi and Lipasti, 2017; Reza et al., 2018; Vengerov, 2009; Yin et al., 2020; Yigitbasi et al., 2013; Braun and Litz, 2019), hence closing the loop of hardware-aided ML algorithms and ML-aided hardware design. Here we go into two specific problems where ML can help human designers in hardware design and optimization.

1.1.1 Controlling Prefetchers at Runtime to Increase Performance

The memory wall is a well-known problem in computer architecture (Wulf and McKee, 1995). The "memory wall" problem, points out that improvement of processor performance far exceeds the improvement of memory speed. Nowadays, because of this trend, the memory bottleneck has become the most important issue in modern computing. To overcome the memory wall, among other approaches like Processing-in-Memory (PIM) (Li et al., 2016b), branch predicting (Smith, 1998), Translation-Lookaside-Buffer (TLB) (Black et al., 1989) Cache Replacement Policies (Al-Zoubi et al., 2004), silicon photonics on chip (Joshi et al., 2009), computer architects have used various prefetching techniques including, Stride (Vanderwiel and Lilja, 2000), Stream (Vanderwiel and Lilja, 2000), and Spatial Memory Streaming (SMS) (Somogyi et al., 2006). Today's processors typically consist of multiple prefetchers. In these systems, there is a contract between the prefetchers and the processor in the form of prefetcher priority, prefetcher throttling, etc., to regulate the interactions between prefetchers. Generally, this contract is tuned in an offline manner for the typical case where all prefetchers are left ON. This tuning can be done at design time regardless of potential program memory access patterns or system conditions in both AMD and Intel CPUs (Intel, 2017; (AMD), 2017). Only tuning the prefetchers for the typical case can cause complex interactions between the prefetchers, which can lead to IPC losses. In systems with multiple prefetchers, several challenges arise from the complex interactions between prefetchers.

Overfitting

Each prefetcher is tuned for a specific type of traffic. For example, a stride prefetcher tracks strided accesses that consist of a single stride and uses thresholds tuned for the average strided sequences. For applications that don't have strided accesses this tuning may be suboptimal, leading to cache thrashing. In a processor with multiple prefetchers, this issue is more pronounced because the architectural resources are shared among all prefetchers.

Duplicate Coverage

In a processor with multiple prefetchers, all prefetchers track the memory patterns. Given all prefetchers train independently, multiple prefetchers may trigger prefetch requests for the same memory patterns. This leads to wasted bandwidth and power consumption.

Faulty Synchronization

Since the prefetchers latch onto memory patterns at different speeds, a prefetcher's behavior over time can be affected by the traffic of the other prefetchers. Variations in the accuracy of each prefetcher and faulty synchronization of the prefetchers with each other can lead to a drop in IPC.

Essentially, prefetchers compete for resources and, at times, sabotage the performance of each other. Instead of turning ON all prefetchers regardless of program phase, one could turn ON only a subset of prefetchers by identifying which prefetchers are best for a given program phase. Leveraging the power of ML to find non-intuitive correlations, we can use

ML to identify when a certain prefetcher should be turned ON.

1.1.2 Securing Against Hardware Trojans Using Nanoantennas

At the heart of security lies the trust established in the underlying silicon that runs the software. Many companies, such as Nvidia, and AMD, only design integrated circuits (ICs) and trust other manufacturers with actually fabricating their products. These companies either have to invest vast amounts of money to fabricate locally (Reuters, 2022; CNBC, 2021b) or they can fabricate their ICs overseas at a much lower cost (Tech, 2019). However, fabricating overseas creates additional attack opportunities for security risks at the fabrication step. These attacks include HT insertion, IC overbuild, reverse-engineering of the intellectual property (IP), side-channel leaks utilizing power and timing, and IC counterfeiting (Karri et al., 2010; Lin et al., 2009; Yang et al., 2016; Jacob et al., 2014; Xiao et al., 2016).

A specific attack gaining recent notoriety is using HTs. HTs are hardware blocks specifically designed to perform attacks on trusted hardware. These attacks can be used to leak information, sabotage or hinder the functions of chips or grant prohibited priority escalation (Becker et al., 2013; Yang et al., 2016). In particular, HTs inserted during the fabrication stage are not easy to detect using standard physical and functional detection techniques (Yang et al., 2016; Zhou et al., 2020; Huang et al., 2020). These HTs have negligible overheads in performance, power, and area, making them extremely difficult to detect (Nowroz et al., 2014; Yang et al., 2016). HTs have even been placed into processors such as the OR1200 processor to gain priority access (Yang et al., 2016). Functional testing is infeasible because these HTs can have uncommon activate sequences (Wei et al., 2012). Current solutions are based on either monitoring the entire manufacturing process or reverse engineering the IC to identify modifications made compared to a known golden design. Reverse engineering involves treating the IC as untrusted hardware and using imaging methods such as Scanning Electron Microscope (SEM) or delayering techniques to detect

HTs (Tehranipoor and Koushanfar, 2010; Karri et al., 2010). Unfortunately, these methods are time-inefficient and have high costs.

Using nanoantennas (Adato et al., 2016; Adato et al., 2015; Zhou et al., 2015; Zhou et al., 2020; Zaraee et al., 2020) to secure IC chips against these fabrication-side HT attacks has been proposed as an alternative to these methods . Nanoantennas are grating pair structures created from metal material that resonate at specific angles of reflectance and optical wavelengths. By inserting nanoantennas into individual standard cells and effectively into the IC chips, we have an optical signature for each IC chip. If the structures within the IC chip are modified, we can observe a change in the optical signature.

While using nanoantennas for optical signature creation is a promising approach, the insertion process of these nanoantennas into standard cells is a design-intensive task. Prior art either made the design task easier by focusing on a few standard cells such as fill cells (Zhou et al., 2020), or they used dual-gate pairs to increase the area to make it easier to insert a nanoantenna (Zaraee et al., 2020). Both approaches are non-ideal. Using only nanoantenna-inserted filler cells, the coverage of the design is not 100% and an attacker could potentially replace the functional cells with a HT. With a dual-gate pair approach, the designs are not readily CAD-flow compliant and require heavy modification of the CAD tools to place gate pairs. Providing complete coverage of the standard cells by inserting nanoantennas into each cell would provide better security guarantees. Additionally, these standard cells would be compliant with existing tools. Therefore, they could easily be used to secure IP readily.

1.2 Thesis Contributions

At a broader level, this thesis explores the idea of using ML to manage and design hardware. By leveraging the power of ML, we find non-intuitive interactions in hardware and optimize the design and management of hardware. We focus on two approaches of using

ML for hardware optimization and design. The thrusts we have presented in this thesis can be extended far beyond the implications shown in this thesis. We envision a future where extremely complicated hardware design choices and management decisions are automatized and controlled by algorithms. In the first approach, we leverage low-overhead ML algorithms to improve the performance of hardware prefetchers. In the second approach, we utilize an RL-based CAD flow to automate the design of secure standard cell libraries. The thesis statement as follows:

Using ML, we can design and manage hardware more efficiently than a human-intuition-backed heuristic approach, leading to better performance and security benefits.

The main contributions of this Ph.D. research are as follows:

- **Puppeteer: An RF-based Hardware Prefetcher Manager -** In the first approach, we propose a novel ML-based runtime hardware manager called Puppeteer to manage the various prefetchers across the memory hierarchy to improve processor performance. Compared to prior work, Puppeteer is the only manager that targets all cache levels in the memory hierarchy. At runtime, during an instruction window, we use a set of invariant events (which can be tracked using hardware performance counters (HPCs)) to count the number of occurrences of various hardware-level events in a processor. At the end of the instruction window, we use these events as input to our ML model, which predicts the prefetcher system configuration (PSC) with the highest IPC for the next instruction window. We use several novel ideas to increase the performance of Puppeteer. First is the utilization of invariant events. Each invariant event is largely independent from the prefetchers' action. We can utilize the independence of these events to prefetching to collect the features and train our model in an offline manner. Next, we train Puppeteer to maximize processor performance instead of the prefetch address prediction accuracy utilizing a regression scheme instead of classification. Finally, we use only 10% of the data for training to prevent overfitting and design Puppeteer with a low hardware overhead ($\sim$10KB). In contrast, prior art does not consider generalization issues and approaches this problem mainly from the perspective of training accuracy. We make sure that Puppeteer is feasible to implement in hardware. Therefore, we co-optimize the hardware design and the ML model of Puppeteer intending to maximize the overall application performance while minimizing the area overhead. For the 232 traces we evaluated, Puppeteer achieves an average performance gain of 46.0% in 1 Core (1C), 25.8% in 4 Core (4C), and 11.9% in 8 Core (8C) processors compared to a system with no prefetching. Moreover, we ensure Puppeteer reduces negative outliers. When using Puppeteer, we observe an 89% reduction in the number of outliers, down to 8 outliers from 53, and a 20% reduction in the IPC loss of the worst-case outlier compared to the state-of-the-art prior prefetcher managers.

- **SecRLCAD: An RL-based CAD flow to Secure Standard Cells -** In the second approach, we propose an RL-based CAD flow to insert nanoantennas into all standard cells in a given standard cell library instead of just non-functional filler cells or dual-gate pairs. In our novel insertion flow, we begin by leveraging a graph-insertion algorithm to insert nanoantennas into existing standard cells as a first pass. We use this as a starting point for a Deep-Q-Network (DQN) RL neural network (NN) to fix DRC issues in the standard cells due to the insertion stage. We automate the flow end-to-end to take an existing standard cell library as input, insert an optical nanoantenna in each cell of the library, and generate a secure standard cell library as output. Our flow is the first of its nature that focuses on the security aspect of a standard cell library. We build the flow to be as generic as possible and to be capable of inserting new nanoantenna designs in the standard cell library of any technology node. We automate the generation of all the standard cell library files such as .lef, .lib, and the .gds files end-to-end with only a few path changes to be compatible with the standard CAD flows. Utilizing SecRLCAD, we can secure FreePDK 45nm at an average area increase of only 56%. Furthermore, for GF22FDX (GlobalFoundries, 2006), we designed secure standard cells for seven min-size standard cell gates and taped out a chip to test our idea of inserting nanoantennas into the functional standard cells. Based on our tape out experience, we can reduce the design time of securing the 100 standard cell gates from 28.5 months to only a couple of hours.

1.3 Organization

The remainder of this thesis is organized as follows. We review the background and state of the art on ML-based prefetching, ML-based hardware management, Hardware IP security, ML-based chip layout algorithms, and optical nanoantenna design in Chapter 2. Chapter 3 presents our work on designing and implementing our low-overhead hardware prefetcher manager called Puppeteer. Chapter 4, introduces our RL-based CAD flow called SecRL-CAD for automatic insertion of optical nanoantennas into standard cell libraries. Finally, in Chapter 5, we discuss the future directions and conclude this thesis.

Chapter 2

Background and Related Work

In this chapter, we provide a brief background on essential areas to Puppeteer and SecRL-CAD. We then provide related work on prior art that is more closely related to this thesis.

2.1 Background

In this section, we provide a background on various topics relevant to the two thrusts we have investigated. In the case of Puppeteer, prefetching and RF are critical topics to understand. In the case of SecRLCAD, we briefly outline the relevant topics in hardware trojans (HT), optical nanoantennas, and RL.

2.1.1 Puppeteer Background

Prefetching

The focus of our work is on a prefetcher manager that can work with any type of prefetcher on any cache level. Our work is also able to manage multiple prefetchers on a single cache level or across cache levels. Broadly, we can split prefetching techniques into several categories. We can have regular-pattern-based, i.e., stride-based prefetchers (Kagi et al., 1996; Wenisch et al., 2009), irregular-pattern-based, i.e., stream-based prefetchers (Cantin et al., 2006; Somogyi et al., 2006), prefetchers that track both regular and irregular patterns (Shevgoor et al., 2015; Kim et al., 2016), and region-based prefetchers (Ganusov and Burtscher, 2005; Wang et al., 2003). An excellent comprehensive survey on prefetchers can be found in Falsafi et al. (Falsafi and Wenisch, 2014).

ML-based algorithms can predict the addresses of the instructions/data that should be prefetched at each cache level in the memory hierarchy. The ML-based solutions include table-based reinforcement learning (Peled et al., 2015; Bera et al., 2021), linear model-based reinforcement learning (Zhang et al., 2018), perceptron-based neural networks (Peled et al., 2019; Fedchenko et al., 2019), Markov chain model (Moreira et al., 2017) and LSTM-based neural networks (Hashemi et al., 2018; Shi et al., 2021b; Zhang et al., 2020a; Narayanan et al., 2018; Srivastava et al., 2019; Braun and Litz, 2019; Zeng and Guo, 2017; Rogers, 2019) to predict memory access patterns. As ML algorithms get more powerful and the techniques to compress these algorithms become more sophisticated, ML-based prefetchers will become commonplace in processors.

Random Forests

RF is a supervised ML Algorithm widely popular in classification and regression problems (Biau and Scornet, 2016). RF is made up of decision trees built on subsets of the training data. The main advantage is since the samples are split up into subsets and used to try and achieve the same outcome, namely minimizing prediction error, the individual decision trees learn the underlying distribution of data from different data. This reduces overfitting issues since a single decision tree with access to all the training data can more closely fit the training distribution. Due to this property, RF has been proven to work quite well with noisy data (Kulkarni and Sinha, 2012). Another advantage of RF is that RF has low computational and area overhead compared to more powerful NN algorithms. A typical NN needs many layers made up of a large number of nodes to achieve high accuracy on noisy data. For the state-of-the-art, NNs can reach billions of nodes (Dai et al., 2021; Riquelme et al., 2021; Brown et al., 2020).

An RF can be formulated as taking the average output of a set of decision trees in the case of regression or taking the majority vote in the case of classification. $f' = \frac{1}{N}\sum_{n=1}^{N} f_n(x)$ where each f_n is a single decision tree function, and N is the total number of decision trees.

2.1.2 SecRLCAD Background

Hardware Trojans

A HT is an intentionally malicious circuit built into an IC. A HT can be used to leak information, disrupt performance, or even destroy an IC given a specific trigger (Hu et al., 2016). The type of a HT can be either functional or parametric. Functional HT are created by adding or deleting any transistors or gates to the original chip design. Parametric HT are created by making circuitry modifications on the original IC, such as changing wire thickness or transistor strength. A HT masks its existence by typically triggering on a sporadic event. A HT can even be inserted during the manufacturing phase and have extremely low area, power, and performance overhead, making it extremely difficult to distinguish the inserted HT (Yang et al., 2016). The rarity of events and the HT's sizing makes HT detection difficult with both functional and physical methods such as scanning electron microscopy (SEM) or physically destructive techniques such as slicing. SecRLCAD provides security against functional HT and parametric HT that modify transistor sizing. SecRLCAD also does not require sophisticated imaging or functional control techniques.

Optical Nanoantennas

Metal in IC chips strongly reflects near-infrared (IR) light while silicon is near-transparent (Zhou et al., 2020). We can then use backside imaging to generate an optical image of the metal layer of the IC (Ippolito et al., 2004; Köklü and Ünlü, 2010). We can construct structures that will create known responses under specific wavelengths and optical angles using this property. These structures are called optical nanoantennas. With nanoantennas, Zaraee et al. have shown that we can generate an optical watermark at the standard cell level (Zaraee et al., 2020). We can then compare a generated finite difference time domain (FDTD) simulation image, "a golden reference", with the actual backside images of

IC to detect mismatches (Zhou et al., 2020). Using nanoantennas to generate an optical watermark is comparatively better compared to SEM imaging since it is faster and does not require expensive machines. Additionally, it is not destructive such as slicing techniques. Zhou et al. used these structures in fill cells to show modifications made to the empty regions of the ICs (Zhou et al., 2020). While Zaraee et al. show that we can insert nanoantennas into dual-gate pairs where two standard cells such as an OR gate and AND gate are placed side by side. Zhou et al. do not provide adequate coverage in the active region of the ICs, while Zaraee et al. do not have a readily available answer to how their dual-gate pairs can be leveraged in existing CAD flows.

Reinforcement Learning

Reinforcement learning (RL) is an algorithmic approach consisting of an RL agent and an environment/state (Sutton and Barto, 2018; Kaelbling et al., 1996; François-Lavet et al., 2018). An RL agent learns how certain actions taken in a given state will affect a reward function. The RL agent then learns to maximize this reward function. A typical RL approach is composed of four critical components:

- **(i) The state vector (S_t) -** The state vector records the current condition of the environment that the RL agent is taking actions on.
- **(ii) The action choice (A_t) -** The RL agent can take several actions upon the state for each time step.
- **(iii) The reward function (R_t) -** For each state, there is a recorded numerical output. The actions taken upon the given state creates a difference in the numerical output. This difference guides the RL agent towards better actions for state pairs.
- **(iv) The stop condition -** The RL agent needs to know when to stop and reset the state vector.

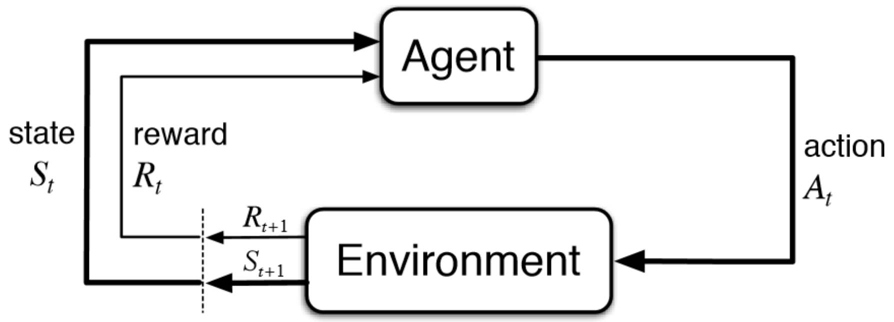


Figure 2·1: Typical RL Algorithm - We have an agent, the environment that it can take actions on and the observed state and reward based on those actions.

This stop can be done on a set number of time steps or when a reward function goal is reached. Figure 2·1, shows the typical interaction between the RL agent and the environment. The RL agent interacts with the state in discrete time steps. At each time step (t) step, the RL agent observes the current state and takes action. Upon receiving the action, the state transitions to a new state at time step ($t+1$) and calculates the reward function, which the RL agent records The reward scheme drives the RL agent towards taking optimal actions.

2.2 Related Work

This section broadly describes the related works to Puppeteer and SecRLCAD. For Puppeteer, we divide the prior prefetcher managers into heuristic-based managers and ML-based managers. For SecRLCAD, we give a more broad description of ML methods used in hardware security as well as ML used for CAD since there is, to the best of our knowledge, no directly related ML-based CAD for security prior work.

Table 2.1: Overview of the Related Work on Prefetcher Managers - We use *NA* when the paper lacks information about the respective metric. In the last column, 'software' indicates the software prefetcher. For IPC gains, we report average IPC gain in 1C. We list the number of cache levels and prefetchers (shortened as Pf) the method manages. We implement and compare against several of these works on our own system to provide a fairer comparison (Liao et al., 2009; Jiménez et al., 2012; Bhatia et al., 2019).

Work	# Pf	# Cache Levels Managed	IPC Gain Over Baseline PSC	IPC Gain Over No Prefetching	Heuristic or ML	Overhead
(Rogers, 2019)	1	1	Only reports accuracy		Heuristic	Software
(Kondguli and Huang, 2018)	1	1	5%	41%	Heuristic	$\sim$ 4.6KB
(Jiménez et al., 2012)	1	1	7.8% (6% from 1 application)	*NA*	Heuristic	Software
(Kang and Wong, 2013)	0	1	11%	No Pf baseline	Heuristic	Software
(Liao et al., 2009)	4	2	Only reports ML accuracy		ML	Software
(Rahman et al., 2015)	1	2	Only reports ML accuracy		ML	Software
(Bhatia et al., 2019)	1	1 Pf targets 2 levels	2.27%	15.24%	ML	$\sim$ 39KB
(Hiebel et al., 2019)	4	1	-1% to 3.6%	-1% to 8.5%	ML	Software
(Maldikar, 2014)	1	1	-5% to 1%	23%	ML	Software
Puppeteer	**7**	**All (4)**	**14.7%**	**46%**	**ML**	$\sim$ 10**KB**

2.2.1 Puppeteer Related Work

Heuristic-Based Prefetcher Managers

When a processor executes an application, there is a diverse set of interactions among a computing system's compute, memory, and communication components. These interactions are dependent on the processor (micro)architecture and the application, thus effectively making the processor a big finite state machine with a vast state space. This makes it challenging to use deterministic techniques, which consider all possible states, for hardware management. As a result, over the past 20-30 years, heuristic algorithms have been used for hardware management, including for prefetcher management such as static priority queue-based approaches (Kondguli and Huang, 2018; Rogers, 2019) and rule-based prefetcher throttling approaches (Ebrahimi et al., 2009a; Ebrahimi et al., 2009b; Ebrahimi et al., 2010; Ebrahimi et al., 2011; Heirman et al., 2018; Hur and Lin, 2006; Srinath et al., 2007; Liu et al., 2016). These algorithms have low memory/area overhead and improve processor performance for the average case. However, heuristic algorithms are no longer effective with the ever-increasing complexity of processors and the diversity of applications. Heuristic algorithms are extremely dependent on the hardware/operating conditions and cannot easily adapt to variations at runtime. But as processors have gotten more complicated and the variety of applications has increased, these algorithms have had difficulty scaling to modern environments and processor designs. They have become much more difficult to design such that they work well with other heuristic algorithms present in the processor.

In contrast, ML algorithms are more adaptable to changing environments and they do not have to be reconstructed from scratch with minor changes in application use cases. Furthermore, ML algorithms have higher versatility and generalization than heuristic algorithms, making them a prime candidate for management decisions.

ML-based Prefetcher Managers

ML methods have been gaining traction in place of heuristic methods for achieving superior prefetcher management performance (Bhatia et al., 2019; Liao et al., 2009; Jiménez et al., 2012; Hiebel et al., 2019; Maldikar, 2014). ML algorithms can extract the non-intuitive interactions between the different prefetchers. Prior methods on prefetcher management configure or train the manager, which typically predicts which PSC to use for a given application, using values of hardware events collected for a single fixed PSC (generally, the default PSC) (Liao et al., 2009; Jiménez et al., 2012; Bhatia et al., 2019; Rahman et al., 2015). Using an ML model trained using only a single fixed PSC would make sense if the prefetcher system always uses that single fixed PSC at runtime. However, at runtime, the PSC changes. If the values of the hardware events are highly dependent on which PSC is being used, using a dataset generated using a fixed PSC for training leads to a low-accuracy ML model for the prefetcher adaptation. To address this concern, we train our ML model using PSC-invariant hardware events (i.e., events that are not dependent on the PSC, e.g., the number of branch instructions).

We observe a wide variation in the complexity of the ML algorithms used in prior work. We list the prior work in Table 2.1. Some of the algorithms are simple, and either use small datasets, or use datasets that do not accurately portray the runtime environment as they use PSC-variant events and only train data of one fixed PSC. As a result, these algorithms cannot achieve good accuracy at runtime (Liao et al., 2009; Jiménez et al., 2012; Rahman et al., 2015). Other algorithms, such as neural networks, are too complex and their size increases prohibitively with the size of the dataset (Bhatia et al., 2019). Moreover, some prior works focus on hardware adaptation only from the perspective of accuracy without worrying about the hardware implementation (Liao et al., 2009; Jiménez et al., 2012; Hiebel et al., 2019; Rahman et al., 2015).

Contrary to the prior work, we jointly account for the accuracy of the ML model and hardware overhead when designing Puppeteer. Puppeteer is complex enough to provide good accuracy on a wide variety of applications. At the same time, Puppeteer is not too complex (as demonstrated in Section 3.2.4) to be implemented in hardware and scales well with the size/complexity of the dataset. Furthermore, Puppeteer is agnostic of the underlying internal mechanics of the prefetchers and can be easily retrained for a new prefetcher that is introduced in a new system. Additionally, to the best of our knowledge, Puppeteer is the only manager that targets all cache levels in the memory hierarchy.

ML-based Managers for Other Hardware Components

Systems for ML has been an active area of research for the past two decades. It is only recently that ML for systems has started to gain a significant amount of traffic (Maas, 2020; Wu and Xie, 2022; Jiménez and Teran, 2017; Shi et al., 2019; Bhatia et al., 2019; Braun and Litz, 2019; Ipek et al., 2008; Mukundan and Martinez, 2012; Calder et al., 1997; Tarsa et al., 2019; Reza et al., 2018; Yin et al., 2020; Cochran et al., 2011; Gomez et al., 2001; Tesauro, 2007; Jain et al., 2016; Cummins et al., 2017). ML has been utilized in a wide variety of hardware components and policies that require runtime decision making and management. ML has been used for cache replacement policies (Jiménez and Teran, 2017; Shi et al., 2019), prefetching (Bhatia et al., 2019; Braun and Litz, 2019), memory controller scheduling policy (Ipek et al., 2008; Mukundan and Martinez, 2012), branch prediction (Calder et al., 1997; Tarsa et al., 2019), link management in NoCs (Reza et al., 2018), NoC arbitration policies (Yin et al., 2020), DVFS and thread management (Cochran et al., 2011), dynamic cache partitioning (Gomez et al., 2001; Jain et al., 2016), job scheduling (Tesauro, 2007), and code generation (Cummins et al., 2017).

2.2.2 SecRLCAD Related Work

ML for Hardware Security

The advent of ML has also brought many benefits to the area of hardware security. ML has been used in networks to detect intrusions and anomalies (Rieck et al., 2011). The prior art has also used ML to evaluate of ciphers against side-channel leaks (Hospodar et al., 2011). Jin et al. detect when a HT has been activated using ML (Jin et al., 2012). Li et al. detect HT using power consumption traces and ML detection techniques for pattern finding in normal and malicious behavior (Li et al., 2016a). Hasegawa et al. use NN to conduct static netlist analysis to detect HT insertion using RF (Hasegawa et al., 2017). Asadizanjani et al. use ML to detect changes in the backside images of ICs (Asadizanjani et al., 2017). Demme et al. have used HPCs to evaluate if a certain application is malicious (Demme et al., 2013). Zhou et al. have interjected that this may not be possible because of the unreliability of the training methods used in some of these works (Zhou et al., 2018). An excellent taxonomy on imaging and ML-based techniques using SEM-based reverse engineering methods can be found in (Botero et al., 2021).

ML for CAD

The area of CAD has been developed over the past four to five decades to push the boundaries of computing. With transistors at the nanometer-scale and the most recent IC designs, such as the Apple M1 Max chip, containing 57 billion transistors (Pugsley et al., 2020a), IC design has become dominated by how fast and effective algorithms can converge to solutions that are both feasible, low-cost, and high-performance. Decades of research have pushed applied mathematics and developed meticulous incremental improvements on algorithms such as simulated annealing. This has primarily been disrupted very recently by a seminal paper by Google (Mirhoseini et al., 2021). n this paper, Mirhoseni et al. show that their RL-based algorithm can achieve chip layouts at the level of the state-of-the-art algorithms that take several days or even weeks amazingly in under six hours. Their algorithm comes up with non-intuitive solutions that an architect could not reasonably come up with. Many CAD companies and research groups have become intensively focused on finding ML solutions for CAD problems with this innovative paper.

Ren et al. use a genetic algorithm with an RL-based DRC fixer to design standard cells from scratch (Ren and Fojtik, 2021; Ren et al., 2021). Ustun et al. use a Graph Neural Network (GNN) algorithm, GraphSage to perform logic synthesis in FPGAs (Ustun et al., 2020). Zhang et al. use a Graph Convolutional Network (GCN) for IC verification and sign-off (Zhang et al., 2020b). Want et al. use a GNN and RL-based flow to more effectively place devices (Wang et al., 2022). Xu et al. use a GCN with an RL agent to perform the first stage of the floorplanning problem as a Markov decision process (Xu et al., 2021a). Jiang et al. improve upon the prior work of Mirhoseni et al. by replacing the initial force-directed placement with DreamPlace (Jiang et al., 2021). Cheng et al. (Cheng and Yan, 2021) show that by combining both Convolutional Neural Networks (CNNs) and GNNs, they can achieve better results for chip placement. Finally, Shi et al. (Shi et al., 2021a) configure an RL agent for a multi-task scenario to perform analog placement.

Chapter 3

Puppeteer: RF-based Low-overhead Hardware Prefetcher Manager

3.1 Introduction

Instruction and data prefetching (Falsafi and Wenisch, 2014) are commonly used in today's processors to overcome the memory wall problem (Wulf and McKee, 1995). The key idea behind prefetching is identifying the current memory access pattern and predicting addresses to proactively fetch instructions and data into the cache to avoid cache misses. Prefetching hides the large memory access latency, and in turn, improves processor performance. As a result, modern processors employ multiple prefetchers to cover a wide range of applications. Consequently, existing prefetchers do not improve the performance of all applications; in some cases they hurt application performance by prefetching the wrong memory addresses (Lee et al., 2012). These incorrect prefetches use up the precious memory bandwidth and the limited space in the cache hierarchy. This increases data and instruction access latency, which hurts application performance.

To evaluate a prefetcher's performance, we can use scope and accuracy as the metrics (Bhatia et al., 2019; Kondguli and Huang, 2018). A prefetcher with high prefetching accuracy usually has limited scope, i.e., it is very good at identifying a limited number of memory access patterns and can accurately prefetch data/instructions if those specific memory access patterns exist. However, such a prefetcher fails to identify other memory access patterns. Conversely, a prefetcher with broad scope caters to a wide variety of memory access patterns, but it has low accuracy.

Effectively, we need to find a balance between the accuracy and scope of the prefetcher. One way to balance scope and accuracy is to use multiple high accuracy prefetchers at each level of the memory hierarchy, as is the case in AMD and Intel processors (Intel, 2017; (AMD), 2017; Bios., 2010; Intel, 2011). Each prefetcher is customized to identify a specific type of memory access pattern and make a prefetching prediction. However, having multiple prefetchers operating at each level in the memory hierarchy can lead to the following issues:

- Given that each prefetcher is trained independently to track a specific type of traffic and simultaneously share microarchitectural resources, prefetchers can sabotage each other during runtime. A prefetcher may trigger prefetch requests that evict cache lines that another prefetcher has accurately prefetched. This behavior leads to a loss in performance, wasted memory access bandwidth, and increased power consumption.

- Different prefetchers (either at the same level or across levels in the memory hierarchy) latch onto memory access patterns at different speeds. So a prefetcher's prediction can be influenced by the traffic generated by other prefetchers. These differences in temporal behavior can cause faulty synchronization among prefetchers and lead to a drop in application performance.

Essentially, prefetchers compete for resources, and at times, sabotage each other. To validate our argument, we use traces[1] generated from SPEC2017, SPEC2006, and Cloud benchmark suites and run these traces on an OoO processor that uses a different prefetcher at each level of the memory hierarchy (details of the particular evaluation methodology

[1]A trace is a group of instructions that represent a specific behavior. One or more traces can be used to represent the behavior of a benchmark. For example, a benchmark with consistent looping behavior can be represented by one trace corresponding to a single iteration of the loop. One or more unique representative traces are generated from each benchmark (Perelman et al., 2003).

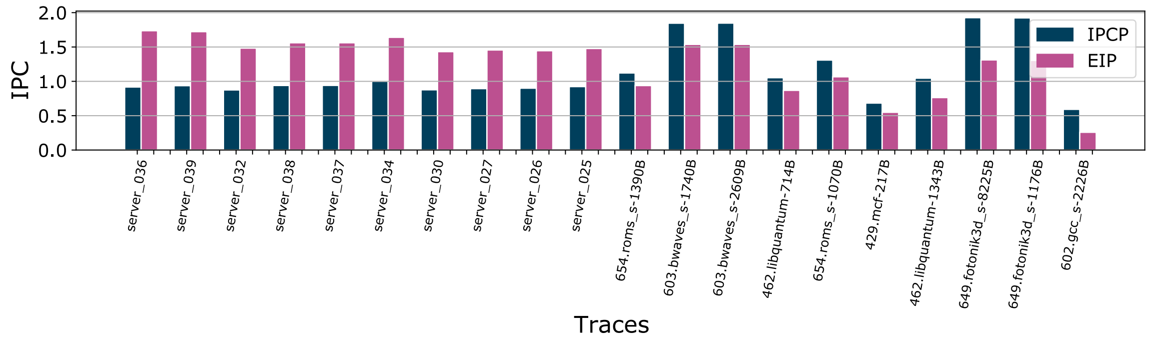


Figure 3·1: Motivational Example - Performance of a processor when using IPCP and EIP prefetchers. Here we show the 20 traces with the highest difference in performance out of 232 traces that we evaluated.

in Section 3.3.1). We execute these traces using two state-of-the-art prefetcher solutions – IPCP (Pakalapati and Panda, 2020) and EIP (Ros and Jimborean, 2020), who were the winners of prior prefetching competitions (Pugsley et al., 2020c; Pugsley et al., 2019; Pugsley et al., 2020b). These prefetcher solutions use different prefetchers at each level of the memory hierarchy, i.e., different prefetcher system configurations (PSC)[2]. Both prefetcher solutions improve performance compared to the no prefetching case. However, IPCP (*no-ipcp-ipcp-nl*) (further details on the types of prefetchers are given in Table 3.4) shows better performance over EIP in 100 out of 232 traces with the largest performance gain of 56% in `602.gcc_s-2226B`. EIP (*EIP-nl-spp-no*) shows better performance over IPCP in the remaining 132 traces and has the largest performance gain of 89% in `server_036`.

In Figure 3·1, we show a subset of traces that have a significant difference in their IPC when using IPCP and EIP prefetcher solutions. The standard methodology would have us select EIP because it has on average 7.8% performance gain over IPCP, but that would come at the cost of having a performance loss for 100 out of the 232 traces.

One way to address this problem is to have multiple different prefetchers at each cache level and switch ON a prefetcher based on the current program phase. For example, one

[2]A PSC specifies which prefetcher is switched ON at each level of the memory in the system. We denote a PSC using the following format `<prefetcher-in-L1I$>-<prefetcher-in-L1D$>-<prefetcher-in-L2$>-<prefetcher-in-LLC$>`.

prior work that has attempted to leverage multiple prefetchers in the *same* level of the memory hierarchy is by Kondugli et al. (Kondguli and Huang, 2018). The authors propose a 'composite prefetcher', which uses a priority queue as the control algorithm to select a prefetcher at a single level in the memory hierarchy. While this approach provides benefits, the 'composite prefetcher' priority queue is designed offline for a given set of applications. For previously unseen applications, we will not necessarily see any performance improvement. Furthermore, the control algorithm will not scale well as we increase the number of prefetchers in the system and target *different* levels in the memory hierarchy.

What is really needed is a *manager* that can successfully manage multiple prefetchers at each level in the memory hierarchy and has a low overhead. This manager will choose the PSC that is suitable for the current phase of an application. The chosen PSC should have prefetchers that complement each other for the current phase, reduce memory access overhead, and in turn improve application performance. Given that multiple prefetchers are available at each level in the memory hierarchy, this 'manager' will effectively determine which prefetcher should be ON/OFF at each level in the memory hierarchy, both across different phases of an application and across applications. **In this chapter we propose a machine learning (ML)-based hardware manager called Puppeteer that selects the PSC at runtime.** Contrary to the prior work (Liao et al., 2009; Zeng and Guo, 2017; Rahman et al., 2015) that focuses on training the ML model to improve the prefetch address prediction accuracy of the ML model, we train the ML model of Puppeteer to increase the overall system performance (quantified as IPC). Using our unique training strategy, we are able to specifically target training for application phases where the swing in IPC is much higher than in other regions. Thereby, we tailor Puppeteer for these phases and achieve high targeted-application-phase accuracy instead of just high overall model prediction accuracy.

To manage the prefetchers across multiple cache levels at runtime, we propose a multi-regression ML-based approach. We use the observed IPC of the various PSCs for different phases of each application to train our ML model. We train a unique random forest regressor per PSC (in our case, we have 5 different PSCs, which we pruned down from the 300 possible PSCs – more details about this in Section 3.3.1), to create a suite of random forests regressors. For features used in the ML model, we use events that can be tracked using hardware performance counters and whose behavior does not change with the choice of PSC, i.e., PSC-invariant events. An example of such an event is the number of branch instructions in an application. The branch instruction count does not change with the choice of PSC. Using only PSC-invariant events, we can limit the number of executions per trace we must account for during training, making it easier to train the ML model in Puppeteer (more details about the training approach are in in Section 3.2.3). In summary, the contributions of our work are as follows:

- We propose a novel ML-based runtime hardware manager called Puppeteer to manage the various prefetchers across the memory hierarchy to improve processor performance.

- We design Puppeteer to use a set of PSC-invariant events (which can be tracked using hardware performance counters) as inputs and predict the PSC for the next instruction window[3]. We co-optimize the hardware design and the ML model of Puppeteer with the goal of maximizing the overall application performance while minimizing the area overhead. At runtime, at the end of an instruction window, Puppeteer predicts the IPC for each PSC and selects the PSC with the highest predicted IPC for the succeeding instruction window.

- We train Puppeteer to maximize processor performance instead of the prefetch address prediction accuracy. We use only 10% of the data for training to prevent overfitting and design Puppeteer with a low hardware overhead ($\sim$10KB). For the 232 traces

[3]An instruction window is group of instructions that are executed sequentially at runtime. We experimented with different instruction window sizes and did not see a large change in the performance of Puppeteer. We set the size to 100,000 instructions.

we experiment with, Puppeteer achieves an average performance gain of 46.0% in 1 Core (1C), 25.8% in 4 Core (4C), and 11.9% in 8 Core (8C) processors on average compared to a system with no prefetching. Moreover, we ensure Puppeteer reduces negative outliers. When using Puppeteer, we observe an 89% reduction in the number of outliers, down to 8 outliers from 53, and a 20% reduction in the IPC loss of the worst-case outlier compared to the state-of-the-art prior prefetcher managers.

- Finally, to the best of our knowledge, Puppeteer is the only manager that targets all cache levels in the memory hierarchy.

3.2 Puppeteer: Design

3.2.1 Puppeteer System-Level Overview

In Figure 3·2, we show the system-level design of an example prefetcher system that uses Puppeteer. The prefetcher system consists of eight different prefetchers (Pf1 to Pf8), which is typical in modern high performance processors such as Intel i9 (Intel, 2017) and AMD Ryzen7 ((AMD), 2017). These eight prefetchers track different memory access patterns and prefetch data from main memory to last-level cache (LLC), from LLC to L2$, and from L2$ to L1$. Pf1 and Pf2 target instruction lines to bring into L1I$, Pf3 and Pf4 prefetch data into L1D$, Pf5 and Pf6 prefetch data into L2$, while Pf7 and Pf8 target data to bring into LLC. These prefetchers compete among them for cache and memory resources. Even across memory levels, a wrong prefetch request from lower levels of memory can harm the performance of prefetchers at higher levels of memory. These prefetchers can sometimes act overly aggressive, and can adversely affect each other, in turn leading to loss of application performance. There are many heuristics-based algorithms that use simple inputs such as accuracy[4] of the prefetchers or memory bandwidth utilization to throttle prefetch-

[4]Here accuracy of a prefetcher is quantified as the fraction of the total prefetches that were actually useful, and is calculated as #prefetches referenced by the program divided by total #prefetches. Note, this prefetcher accuracy is not the same as the ML model accuracy. Scope is calculated as total #misses eliminated by the

ers in such adverse scenarios (Ebrahimi et al., 2009a; Ebrahimi et al., 2009b; Ebrahimi et al., 2010; Ebrahimi et al., 2011; Heirman et al., 2018; Hur and Lin, 2006; Srinath et al., 2007). These heuristic algorithms are designed to have low overhead, and hence are highly optimized for the prefetchers in a given prefetcher system.

Puppeteer works as a manager of all prefetchers and complements the heuristic algorithms used in the given prefetcher system. At runtime, Puppeteer periodically updates the PSC, i.e., it sets which prefetcher should be ON and which should be OFF at each level in the memory hierarchy. To update the PSC, Puppeteer uses an ML model with the PSC-invariant hardware events, collected from hardware performance counters (HPCs) as inputs. While low-overhead heuristic algorithms are still required to make extremely low latency decisions at the cycle level, Puppeteer provides additional adaptability by leveraging the power of ML and thereby increasing the performance. Effectively, heuristic algorithms such as throttling are used in the system to constantly regulate the short-term behavior of the prefetchers, while Puppeteer controls the longer-term system-level behavior (across hundreds of thousands of cycles).

3.2.2 Puppeteer Algorithm

For the ML-based Puppeteer algorithm we considered a classification-based approach and a regression-based approach. The classification-based approach has been used by prior works because it is relatively easy to train offline and has lower hardware overhead compared to regression. The regression-based approach has potential for higher performance and has better tolerance to variability compared to classification during runtime when trained offline.

Classification vs. Regression: To train Puppeteer, as a classification problem we created a dataset using thresholding method similar to prior works (Liao et al., 2009; Jiménez et al., 2012; Bhatia et al., 2019; Maldikar, 2014). Here, a trace is run using all available PSCs. A

prefetcher divided by the total #misses when prefetching is disabled.

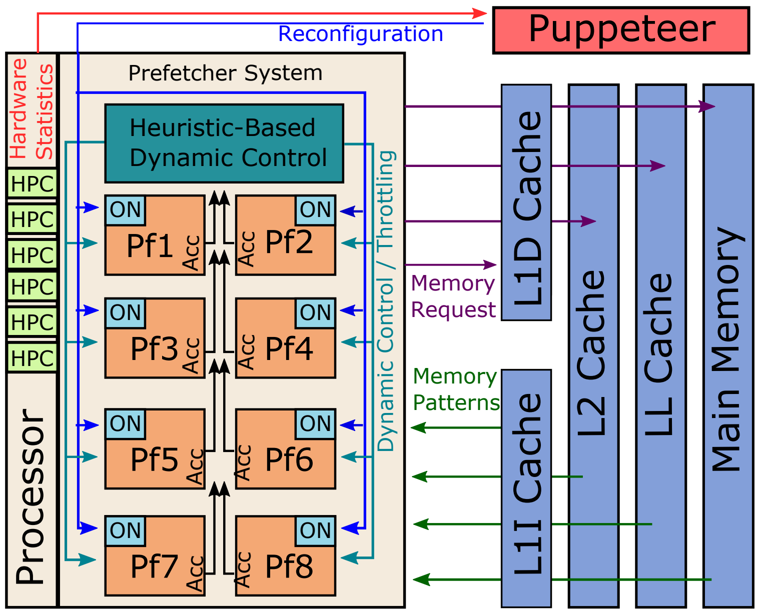
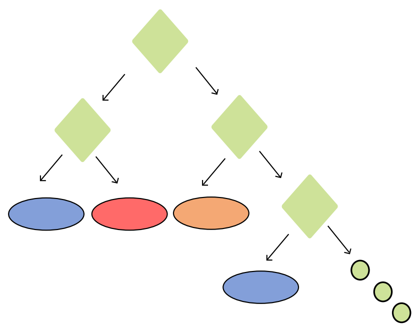


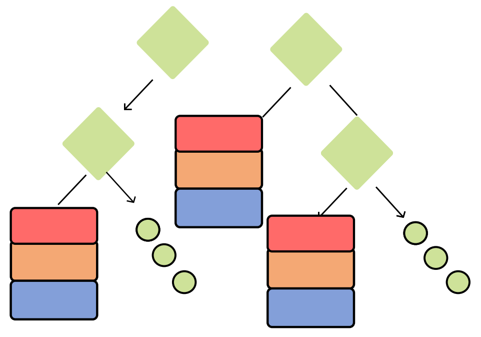
Figure 3·2: Overview of a Puppeteer-based System - Here Pf = prefetcher. Acc = Accuracy of the prefetchers. Pf1 and Pf2 target instructions to bring into L1I\$, Pf3 and Pf4 prefetch data into L1D\$, Pf5 and Pf6 prefetch data into L2\$, while Pf7 and Pf8 target data to prefetch into LLC. Heuristic-Based Dynamic Control block, is a heuristic algorithm that controls the low-level cycle behavior of the prefetchers. Puppeteer controls the longer-term behavior. HPC values are fed into Puppeteer as inputs at runtime.

PSC is given a label of “1” if the IPC when using that PSC is within some threshold (in the case of the prior work the threshold is 0.5%) of the IPC when using the ideal PSC. Multiple PSCs can pass the chosen threshold for a given program phase and we then end up using the PSC that is predicted as “1” with the highest probability. Otherwise, the PSC receives a label of “0”. Using such a classification approach leads to sub-optimal results.

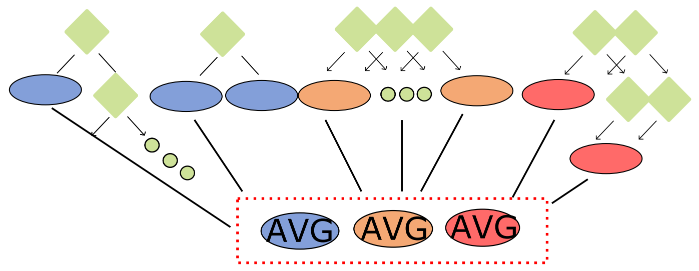
As an example, consider we have four different traces. Let us say we classify the first three out of the four traces correctly and the fourth one incorrectly – i.e. we are able to identify the correct PSC for the first three traces, but not the fourth trace. So, our classification accuracy is 75%. This means that the first three traces will have performance that is within 0.5% of their ‘ideal’ performance. However, the performance of the fourth trace could be 100% worse or just 0.51% worse than the ‘ideal’ performance. This variation in the performance is not accounted in the classification-based model. Furthermore, this



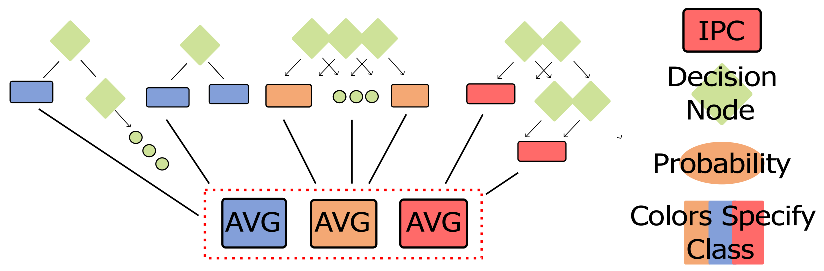
(a) Single Classifier



(b) Single Regressor



(c) Suite of Classifiers



(d) Suite of Regressors

Figure 3·3: Puppeteer Algorithm Options - A decision tree for a singular classifier, a random forest for singular regressor, a suite of decision trees for a suite of classifiers, and a suite of random forest regressors for a suite of regressors, i.e. Puppeteer.

problem is not unique to the given example. Any classification-based method would have a similar issue because all labeling methods used to create the dataset for the classification algorithm will certainly lose some amount of information.

In contrast, a regression-based approach accounts for the *value* of IPC gain/loss and not just if there is IPC gain/loss, when deciding the PSC. Given that the regression algorithm is trained on the IPC values directly, the quantitative information of IPC gain/loss is not lost, and the regression algorithm can learn the magnitude of a good or bad prediction. In particular, the regression algorithm will be used to predict an IPC value for each PSC for a given instruction window. Then, we choose the PSC with the highest predicted IPC value.

Suite of Regressors vs Single Regressor: When using regression algorithms, we have two options: (i) use a single regressor, where all data collected from all the PSCs are used to train that single regressor; or (ii) use a suite of regressors, where each PSC will have a dedicated regressor. Using a suite of regressors leads to a more customized solution that has higher accuracy as compared to using a single regressor. Conceptually, this is because in a single regressor, we maximize the accuracy across all PSCs instead of maximizing the accuracy of each PSC separately, whereas, in a suite of regressors we train a dedicated regressor for each PSC. In this way, we indirectly jointly increase the scope and accuracy of the overall prefetching system.

In our work, we use a suite of regressors where we implement each regressor using random forest, i.e. one random forest trained per PSC, due to its simple implementation, its robustness to noise in the dataset, its lower overhead (compared to other ML algorithms such as neural networks), and its higher accuracy (compared to other ML algorithms such as decision trees) (Kursa, 2014; Oshiro et al., 2012).

We have multiple trees per forest and we allow each tree to split at locations that are unique to the PSC associated with the forest. The leaves of each tree in the forest specify the predicted IPC value for the PSC. For each forest i.e. each PSC, we calculate the average of the predicted IPC values obtained from all the trees in the forest, and then choose the PSC with the highest average predicted IPC. Given that each forest has multiple decision trees, our method has higher tolerance to wrong decisions by the trees, where even if some of the trees give wrong decisions, other trees can compensate.

In Figure 3·3, we conceptually show the differences between the four different options discussed above: (a) single classifier, (b) single regressor (c) a suite of classifiers, and (d) a suite of regressors. Respectively, the leaf nodes in (a) contain the PSC choice directly, (b) have the predicted highest IPC among all the PSCs, (c) the probability value of a given instruction window belonging to the given PSC (of which we choose the highest one), and (d) the predicted IPC value for a given PSC which will be averaged per tree in a given RF and compared with the other averaged predicted PSC values from the other RFs (of which we will choose the highest). For (a) and (b) the PSCs are traversed simultaneously, since the PSCs share a single algorithm, while for (c) and (d) each PSC has a unique algorithm we traverse.

3.2.3 Puppeteer Training

To train Puppeteer we need to generate a representative dataset. Consider the case where we have a single prefetcher, Pf, at only one level in the memory hierarchy. Here the number of PSCs $(N_{psc}) = 2$, i.e., Pf=OFF and Pf=ON. For two consecutive instruction windows, we will have $N_{psc}^2 = 4$ possible scenarios: (i) Pf=OFF $\rightarrow$ Pf=OFF, (ii) Pf=ON $\rightarrow$ Pf=OFF, (iii) Pf=OFF $\rightarrow$ Pf=ON, and (iv) Pf=ON $\rightarrow$ Pf=ON. With N number of instruction windows and N_{trace} number of traces, the number of different possible scenarios will then be $N_{trace} \times N_{psc}^N$. When N increases, the number of different scenarios will increase exponentially, hence including each unique scenario in the dataset for training is not feasible. To handle this problem, we propose to use only PSC-invariant events as our features. An example of a PSC-invariant event is the number of conditional branches, which is not affected by the choice of PSC. We check the variance of each hardware event value (for

180 total hardware events that we can track) for each PSC. We identify 59 events whose values vary by less than $\pm 10\%$ from their mean value across all PSCs. We further reduce the number of events by eliminating the redundant events that track similar behavior and have high correlation with each other. Table 3.1 shows the final 6 events we choose to track trace behavior. After we have identified our PSC-invariant events that will be the features and the PSCs that will be the choices of our ML model, we collect an IPC value per PSC for each instruction window as our ground truth.

Using the features and IPC values we have collected, we then form our suite of random forest regressors wherein we train a separate forest for each PSC using CART (classification and regression trees) (Steinberg, 2009). CART is a greedy recursive search algorithm that maximizes information by splitting the data at each node using one feature. Each child node is split recursively until there is no information gain from splitting a child node. We limit the total number of decision nodes in Puppeteer to keep the size of Puppeteer smaller than L1$. With this limitation in mind, we conduct a hyper-parameter search and determine that the number of estimators (trees per random forest) should be 5 and the number of max nodes should be 100 per tree.

3.2.4 Puppeteer Microarchitecture Design

Figure 3·4 shows the microarchitecture details of Puppeteer. We use a single port SRAM array called *Node MEM* to store information about the nodes that form the trees of each random forest in Puppeteer. We load the random forest-based Puppeteer model into the *Node MEM* at startup using firmware. Each entry of *Node MEM* corresponds to one node in one of the random forests and it consists of the following fields: (i) A 3-bit `HPC ID` field that specifies which PSC-invariant event, i.e. which hardware performance counter (HPC), is used by that node to make a decision. The 3-bit encoding enables the node to use one of 6 different PSC-invariant events (see Table 3.1). (ii) A 16-bit `Threshold` field (threshold value is determined during training), which is employed by the node to

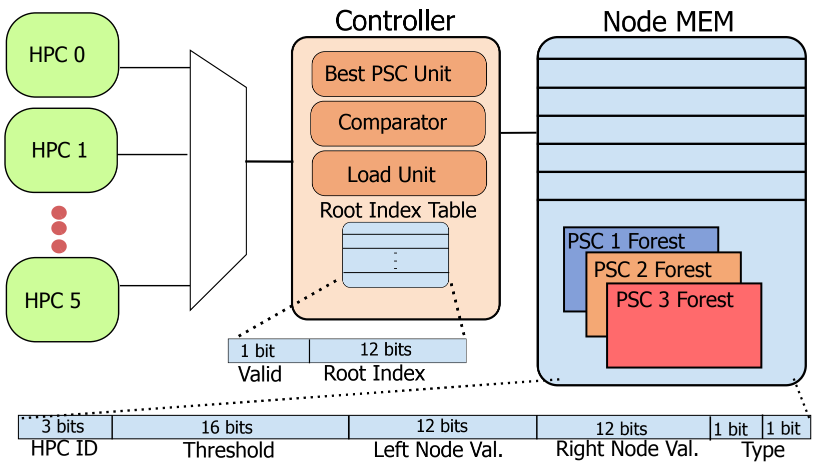


Figure 3·4: Puppeteer Hardware Design - Puppeteer is made up of a Node MEM - SRAM array, a max logic unit, and several register files.

decide if the decision path should branch left or right. In our problem 16 bits provide enough precision for the ML model weight values. (iii) A 12-bit (for 2250 node addresses) `Left Node Value(LNV)` field, and (iv) a 12-bit `Right Node Value(RNV)` field. These `LNV` and `RNV` fields represent child node indices for internal nodes of a tree. For the leaf nodes of a tree, we use these `LNV` and `RNV` fields to indicate the predicted IPC value of a PSC. We differentiate between child node index and predicted IPC using (v) a 1-bit `Type` field. We use a separate 1-bit `Type` field for `LNV` and `RNV`.

At the end of every instruction window, Puppeteer calculates the predicted IPC for each PSC in the next instruction window by traversing the trees of the associated forest and using the PSC-invariant event values for the current window as inputs. For each forest, the controller in Puppeteer reads the *Node MEM* index of the root node for the first tree from *Root Index Table (RIT)* and loads the *Node MEM* entry for the root node using a *Load Unit* into a register. Next, the `HPC ID` in the loaded *Node MEM* entry is used to load the corresponding PSC-invariant event value into a second register. Then the `Threshold` value, stored in the first register, and PSC-invariant event value stored in the second register are compared using the *Comparator*. Based on the *Comparator* output, we choose to traverse down to the left child or the right child. The *Controller* then uses the corresponding index value from `LNV` or `RNV` to find the next node in *Node MEM*. The *Controller* continues traversing the tree until it loads a predicted IPC value corresponding to a leaf from the *Node MEM*. The above steps are repeated for the remaining trees in the forest, and then we calculate the average of the predicted IPC values obtained from all the trees in that forest. The *Best PSC Unit* in the *Controller* stores the ID of the PSC with the highest predicted IPC value. Every time the *Controller* finishes traversing a forest, the predicted IPC value of that forest, i.e. PSC, is compared with the predicted IPC value stored in the *Best PSC Unit* using the *Comparator*. If the new predicted IPC value is higher than the current value, the *Best PSC Unit* updates the predicted IPC value and the ID of the PSC. Once all forests have been traversed i.e., all PSCs have been evaluated, Puppeteer chooses the entry stored in the *Best PSC Unit* as the PSC for the next instruction window.

We determined that a maximum depth of 10 per tree is more than sufficient to accurately determine the best PSC. In our evaluation we use a prefetcher system with N_{psc}=5 (given in Table 3.5 and discussed in detail in Section 3.3.1). We need a total of 2250 nodes to design the trees in Puppeteer, and these nodes require a 10.75 KB-sized *Node MEM* (compared to a typical L1$ of 32 KB). Other than *Node MEM*, we require a $5\times\mathrm{N}_{psc}$-entry *RIT* where each entry is 13-bit wide (12 bits for the root node index and 1 valid bit), a 12-bit comparator containing comparison logic and two registers, a load unit, and a register to store the best PSC information in the *Controller*. We discuss the hardware overhead in more detail in Section 3.3.2.

3.3 Evaluation Framework

3.3.1 Evaluation Methodology

We use ChampSim (Pugsley et al., 2020b) for our analysis, where we model one core (1C), four core (4C), and eight core (8C) processors to have multiple prefetchers at each level of the cache – private L1I$, L1D$, private L2$ and shared LLC (more details provided

Table 3.1: Simulated System Parameters

Component	Simulated Parameters
Core	One to four cores, 4GHz, 4-wide, 256-entry ROB
TLBs	64 entries ITLB, 64 entries DTLB, 1536 entry shared L2 TLB
L1I$	32KB, 8-way, 3 cycles, PQ: 8, MSHR: 8, 4 ports
L1D$	48KB, 12-way, 5 cycles, PQ: 8, MSHR: 16, 2 ports
L2$	512KB, 8-way, 10 cycles, PQ: 16, MSHR: 32, 2 ports
LLC	2MB/core, 16-way, 20 cycles, PQ: 32×cores, MSHR: 64×cores
DRAM	4GB 1 channel/1-core, 8GB 2 channels/multi-core, 1600 MT/sec

Hardware Event	Properties
L1I_PAGES_READ_LOAD	L1I$ Pages Read on Load.
L1D_PAGES_READ_LOAD	L1D$ Pages Read on Load.
L1D_RFO_ACCESS	L1D$ Store Accesses.
BRANCH_RETURN	Branch Returns.
NOT_BRANCH	Not Branches.
BRANCH_CONDITIONAL	Conditional branches.

Table 3.2: Static PSCs and Managers Evaluated - We list the static PSCs from the prior prefetching competitions, the manager algorithms from prior work, and the different flavors of Puppeteer.

Algorithm Notation	Explanation	Static PSC or Manager	Training Dataset
NO	No prefetching is used. PSC = *no-no-no-no*	Static PSC	Not trained
IPCP	Winner of DPC3 (Pugsley et al., 2019). PSC = *no-ipcp-ipcp-nl*.	Static PSC	Not trained
EIP	Winner of IPC1 (Pugsley et al., 2020c). PSC = *EIP-nl-spp-no*.	Static PSC	Not trained
PY	RL-based algorithm named Pythia (Bera et al., 2021)	ML-based Prefetcher	Online
J3	Heuristic-based algorithm by Jimenez et al. (Jiménez et al., 2012)	Manager	Not trained
NN	Multi-layer-perceptron similar to Bhatia et al. (Bhatia et al., 2019)	Manager	1C
B1C	Decision tree algorithm by Liao et al. (Liao et al., 2009)	Manager	1C
B4CS	Decision tree algorithm by Liao et al. (Liao et al., 2009)	Manager	4CS
B4CM	Decision tree algorithm by Liao et al. (Liao et al., 2009)	Manager	4CM
P1C	Puppeteer	Manager	1C
P4CS	Puppeteer	Manager	4CS
P4CM	Puppeteer	Manager	4CM

in Table 3.1). We train Puppeteer using data collected from a 1C and 4C OoO processor and then evaluate it on 1C, 4C, and 8C OoO processors. In the 4C and 8C processors, we have a private Puppeteer per core. Each private Puppeteer utilizes the 6 PSC-invariant events per core and makes independent decisions. For our evaluation we use a diverse set of 232 traces generated from SPEC2017 (Corporation, 2017), SPEC2006 (Corporation, 2006), and Cloud (Pugsley et al., 2020c) benchmarks.

In Table 3.2 we list the notations used for all the static PSCs and the runtime managers (that change PSC at runtime) that we have evaluated. In our evaluation, we normalize all IPC values to the same state-of-the-art baseline as PPF (Bhatia et al., 2019), i.e., SPP (Kim et al., 2016) (*no-no-spp-no*). We compare Puppeteer against the best static PSCs from competitions, i.e., IPCP (*no-ipcp-ipcp-nl*) and EIP (*EIP-nl-spp-no*; as well as managers such as the final version of the algorithm developed by Jimenez et al. that uses trial periods to latch onto the PSC (Jiménez et al., 2012) (J3), Pythia that is an RL-based algorithm developed by Bera et al. (Bera et al., 2021) (PY)[5], a multi-layer-perceptron similar to Bhatia et al. (Bhatia et al., 2019) (NN), and a binary-tree (BT) based algorithm (Liao et al., 2009) - where Liao et al. tried several different ML methods (such as decision trees and NN) and concluded that decision trees are the best choice.

Training Approaches for ML-based Prefetcher Managers

When evaluating any new idea, a widely used approach in the industry is to use all available evaluation data. We are using a ML-based approach and are cognizant of the fact that we do not want to overfit our model using all available data for training the ML model. So here, we compare three different approaches for constructing the training dataset for Puppeteer. For training, we have 232 traces$\times$10,000 instruction windows per trace = 2,320,000 instruc-

[5]Note that Bera et al. used SHiP (Wu et al., 2011) for their cache replacement policy and perceptron as their branch predictor. We tested with their settings as well as with using Pythia with hashed-perceptron and LRU, and observed on average that Pythia with hashed-perceptron with LRU has 12% higher IPC. Hence, we use hashed-perceptron and LRU similar to all the other comparison points in this paper.

tions windows. For each approach, we limit the training data in different ways. Note that we are still training at the instruction window-level and performing 10-fold cross-validation on the training set for all approaches. We give the percentage of data used to construct each training dataset in Table 3.3.

In the **first approach**, we severely limit the data and randomly select 10% of the instruction windows at random time intervals to form the training dataset, i.e., 90% of the instruction windows are not seen during training. The idea behind this approach is that we are training the algorithm to recognize low-level memory access behavior. This approach does not overfit the model because the behavior of the benchmarks when collecting the training data would be different from the behavior of the benchmarks when we use the trained Puppeteer. This difference is because when collecting training data we do not change the PSC, while when using Puppeteer the PSC can potentially change for each instruction window. In the **second approach**, we generate the dataset by leaving 20% of the traces out of the training dataset and using 60% of the instruction windows from the remaining 80% traces for training. We use a 60-40 split of the instruction windows to avoid overfitting the model to the 80% of traces in the training set. Given traces are constructed to represent unique behaviors in each application, our training set will not have information on all the unique behaviors of a given benchmark. This means that some of the traces from other benchmarks are assumed to be from the same distribution of these missing traces or else there will be no way for the algorithm to train for these regions. In the **third approach**, we generate the dataset by leaving 20% of the benchmarks out of the training dataset and using a 60-40 split of the instruction windows belonging to the remaining 80% benchmarks for training. In this approach, whole benchmarks are not considered during training. Here too, we choose a 60-40 split of the instruction windows to avoid overfitting the model to the 80% of benchmarks in the training set. Similar to the second approach it is assumed that some of the other benchmarks will be similar in behavior to the missing benchmarks and cover their memory access behavior.

In a 1C processor, when using Puppeteer on unseen benchmarks, for the first, second, and third approaches, we observe an average IPC gain of 11.3%, 9.8%, and 6.6%, respectively, over SPP. In the second and third approaches, we see that Puppeteer has lower performance gain than the first approach. Despite having 3% more instruction windows for training in the third approach compared to the second approach there is a 3.2% drop in performance. Furthermore, in the first approach we use only 10% of the total number of instruction windows for training (5 $\times$ less instruction windows than the other approaches), yet this approach performs 1.5% and 4.7% better than the other two approaches. This implies that the second and third datasets is trained on an insufficient number of unique memory accesses patterns, leading to low performance observed during runtime execution. Therefore for the rest of our evaluation we train all ML-based prefetcher managers using the first approach. As mentioned earlier, Puppeteer can always be retrained and updated via firmware.

1C, 4C and 8C Workload Formulation

We run 1C and 4C experiments using a 200M instruction warmup phase and 1B instruction detailed simulation phase, while for 8C we use 200M instruction warmup phase and 250M instruction detailed simulation phase. We generate a total of 232 traces from SPEC2017, SPEC2006, and Cloud benchmarks. We create two flavors of trace sets for the 4C and 8C experiments: a single-type trace set where each core runs the same trace, and a mixed-type trace set where each core runs a unique trace. Therefore we have 5 data suites in total: 1C, 4C single-type trace set (4C STS), 4C mixed-type trace set (4C MTS), 8C single-type trace set (8C STS), and 8C mixed-type trace set (8C MTS). We use hashed-perceptron for branch predictor and least-recently used (LRU) policy for cache replacement policy provided by ChampSim. To construct our mixed-type trace sets, we first split the 232 traces into 6 groups based on ascending execution latency. Then we randomly select a trace from a given group per core and construct a mixed-type trace group. We cover all permutations of the various latency groups. This way we have diversity in the traces running on the cores. For example, in a 4C processor, we can assign a trace from group 2 to core0, a trace from group 3 to core1, a trace from group 4 to core3, and a trace from group 5 to core3. This can be represented as $2-3-4-5$. Therefore we have 360 experiments (6 options for core0 $\times$ 5 options for core1 $\times$ 4 options for core2 $\times$ 3 options for core3) in 4C MTS for 6 latency groups and 4 selected traces (1 for each core). For 8C MTS, since the number of experiments increases and the experiments take quite long, we replicate the same latency group for 4 of the cores. For example, we can use $1-1-1-1-4-4-4-4$ for an 8C experiment. However, instead of 6 groups as in the 4C system, we use 10 groups for the 8C system, to cover a more granular variety of behavior and instead of permutations we use combinations with replacement. Therefore, we have 55 (2 traces selected from 10 latency groups with replacement, i.e., $C^R(10,2)$) experiments for 10 groups and 2 selections. Note that, even if the latency group is the same, since we choose a trace randomly from a latency group, the trace can still be different for each core. After we have our experiments constructed, we run each trace on its assigned core until completion. If a trace finishes before all traces have finished, we rerun it from the start. Therefore, each trace will have run at least one time until completion. Compared to the prior work our MTS construction methodology covers a wider variety of behavior. The prior work takes two different approaches to multi-core evaluations. Either they evaluate using only STS workloads (Liao et al., 2009; Bera et al., 2021) or they construct completely random MTS workloads that do not guarantee a wide variety of memory behavior (Bhatia et al., 2019; Jiménez et al., 2012).

Generating ML Models and PSC Pruning

To generate the 1C dataset (PSC and associated IPC values), we run the 1C experiments using all possible PSCs generated from the prefetcher options that are available in the

Table 3.3: Dataset Training Approaches - Here % Benchmarks indicates the percentage of benchmarks were used during training, % Traces indicates the percentage of traces from the available benchmarks were used during training, and % Inst. Win. indicates the percentage of instructions windows from the the available traces were used during training. The last columns indicates overall the percentage of instruction windows were using during training.

Approach	**%Benchmarks**	**%Traces**	**%Inst. Win.**	**Total % of all Inst. Win. for training**
# 1	ALL	ALL	10%	10%
# 2	ALL	80%	60%	48%
# 3	80%	ALL	60%	51%

Table 3.4: Initial Prefetcher Options - We show the regular and irregular prefetcher options we used at each cache level to construct our set of 300 PSCs. We reduce the number of PSCs down to 5 PSCs before training.

L1I$	**L1D$**	**L2$**	**LLC**
No Prefetcher	No Prefetcher	No Prefetcher	No Prefetcher
Next-line (Falsafi and Wenisch, 2014)	Next-line	Next-line	Next-line
FNL+MMA (Seznec, 2020)	IPCP (Pakalapati and Panda, 2020)	IPCP	
DJOLT (Nakamura et al., 2020)	MLOP (Shakerinava et al., 2019)	SPP (Kim et al., 2016)	
EIP (Ros and Jimborean, 2020)	Bingo (Bakhshalipour et al., 2019)	KPCP (Kim et al., 2017)	
		Ip-Stride (Falsafi and Wenisch, 2014)	

Table 3.5: PSCs and Their Prefetching Options used at each $ Level - Note, these PSCs are used by all the manager algorithms not just Puppeteer.

Final PSCs Used	**L1I$**	**L1D$**	**L2$**	**LLC**
djolt-bingo-nl-nl	DJOLT (Nakamura et al., 2020)	Bingo (Bakhshalipour et al., 2019)	Next-line (Falsafi and Wenisch, 2014)	Next-line
djolt-bingo-no-no	DJOLT	Bingo	No Prefetcher	No Prefetcher
fnl-bingo-spp-nl	FNL+MMA (Seznec, 2020)	Bingo	SPP (Kim et al., 2016)	Next-line
fnl-bingo-spp-no	FNL+MMA	Bingo	SPP	No Prefetcher
no-nl-spp-no	No Prefetcher	Next-line	SPP	No Prefetcher
Overhead	221KB	48.06KB	6KB	0.6KB

ChampSim repository, the 1st place (IPCP (Pakalapati and Panda, 2020)), 2nd place (Bingo (Bakhshalipour et al., 2019)), and 3rd place (MLOP (Shakerinava et al., 2019)) winners of the 3rd data prefetching competition (DPC3) (Pugsley et al., 2019), and the 1st place (EIP (Ros and Jimborean, 2020)), 2nd place (FNL+MMA (Seznec, 2020)), and 3rd place (DJOLT (Nakamura et al., 2020)) winners of the 1st instruction prefetching competition (IPC1) (Pugsley et al., 2020c). To reduce the hardware overhead of Puppeteer, we avoid including multiple PSCs that cover the same traces. To this end, we initially ran 20 traces for 20M instructions with all possible PSCs (5 prefetching options in L1I$ $\times$ 5 prefetching options in L1D$ $\times$ 6 prefetching options in L2$ $\times$ 2 prefetching options in LLC = 300 PSCs). We summarize the different prefetchers that we evaluated in Table 3.4.

For each trace, we sort the PSCs based on the corresponding IPC values in descending order. We generate a new table for each trace, where the table contains the top 10 PSC entries for the trace and combine these tables to form a super-table that contains the top 10 PSCs for all traces. Note that a PSC may be in the top 10 for more than one trace. We sort the PSCs in descending order based on the number of traces for which the PSC is in the top 10. Starting from the top, we select just enough PSCs to improve performance of all the 20 traces. We picked the PSCs that have the best performance with minimal coverage overlap while reducing the number of unique prefetchers (to reduce the hardware overhead). In

Table 3.5 we show the final 5 PSCs that we selected. These PSCs give good performance for the maximum number of traces. We show the prefetcher used at each cache level. It is interesting to note that the best Prefetchers from DPC3 and IPC1 – IPCP and EIP – are not used in the top 5 choices for PSCs. This shows that the state-of-the-art prefetchers designed in isolation may not be the best choice of prefetchers when used with other prefetchers.

We collect the 4C STS and 4C MTS datasets, in the same way as the 1C dataset – the only difference is that the data is collected per core. We then train BT and Puppeteer using datasets collected from 1C, 4C STS, and 4C MTS. We denote the different flavors of the two algorithms as P1C, P4CS, and P4CM for Puppeteer; and B1C, B4CS, and B4CM for BT.

Secondary Metrics for Evaluation

In this work our primary metric is IPC. Contrary to the prior work, Puppeteer is not trained to increase the ML model accuracy but to directly maximize the IPC. We do have a number of secondary metrics we track to understand the underlying phenomena that is causing the IPC increase. For the prefetchers, we track prefetching accuracy and scope of the prefetchers. The relationship between prefetching accuracy and scope; and IPC is more nuanced than a linear correlation. This is because the temporal effects of a small number of prefetches might have a larger impact compared to a large number of prefetches. Take for example a single memory line that if prefetched in a timely manner, prevents a whole network from being deadlocked and flushed. Hence, these metrics give us some insight into the average behavior of the prefetchers but do not provide a complete picture. Hence, the reason why IPC is a more direct metric into the impact of Puppeteer on the prefetchers. Our evaluation for prefetching accuracy and scope can be found in Section 3.3.2.

Next, we investigate a variety of trade-offs to get a more complete picture of the solutions space. We investigate the impact of hardware area overhead and power by changing the size of the ML models and retraining Puppeteer and NN. We try to evaluate the impact of the hardware size as realistically as possible by designing the hardware components using a commercial process (GF22FDX). Thereby, we calculate realistic power, delay, and area overhead values. The evaluation for area and delay can be found in Section 3.3.2 and the evaluation for power can be found in Section 3.3.2. Finally, we also wish to test the generalizability of Puppeteer. Although this is hard to quantify we can test for this by changing the underlying hardware or dataset that is used to train Puppeteer. To test for the generalizability, we vary the size of the caches to stress-test the adaptation capabilities of Puppeteer. This study can be found in Section 3.3.2. We also change the dataset used to train Puppeteer and BT and show these results all throughout Section 3.3.2.

3.3.2 Evaluation Results

Puppeteer in a 1C Processor

In this section, we present the processor performance analysis when using Puppeteer in 1C, 4C and 8C processors, a sensitivity analysis of how Puppeteer's performance varies with cache size, and we explore the trade-off between performance improvement and hardware overhead when using Puppeteer. We evaluate Puppeteer using scope, accuracy, and power metrics. For the analysis presented in Sections 3.3.2, 3.3.2, 3.3.2, and 3.3.2, we assume a total hardware overhead budget of 10KB for storing the Puppeteer model, and due to its simplicity (as explained in Section 3·2), we assume negligible evaluation overhead for the computing logic.

In Figure 3·5, we show the IPC distribution for various static PSCs and prefetcher managers that change the PSC at runtime. We begin our discussion with P1C which is Puppeteer trained using the 1C data suite. Broadly, compared to a processor with no prefetchers, P1C provides an average performance gain of 46.0% (peak value of 613%). The true benefit of Puppeteer is observed when we look closer into the performance loss in Figure 3·6. We observe that when using B1C, 53 traces lose performance and 6 out of those 53 traces lose more than 10% performance. This performance loss would make this solution

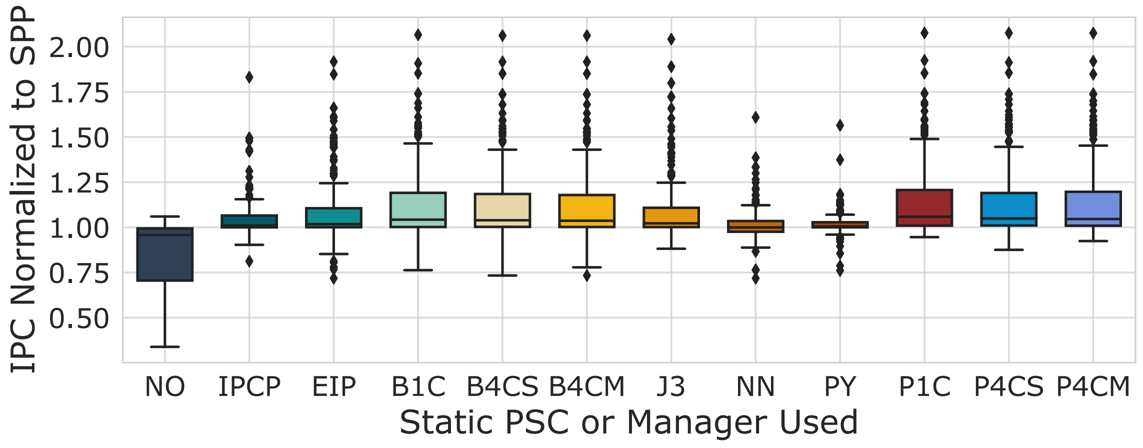


Figure 3·5: Normalized Performance of 1C Processor - Performance distribution of Puppeteer and prior work normalized to *SPP* (Kim et al., 2016). See Table 3.2 for the notations used along the X-axis.

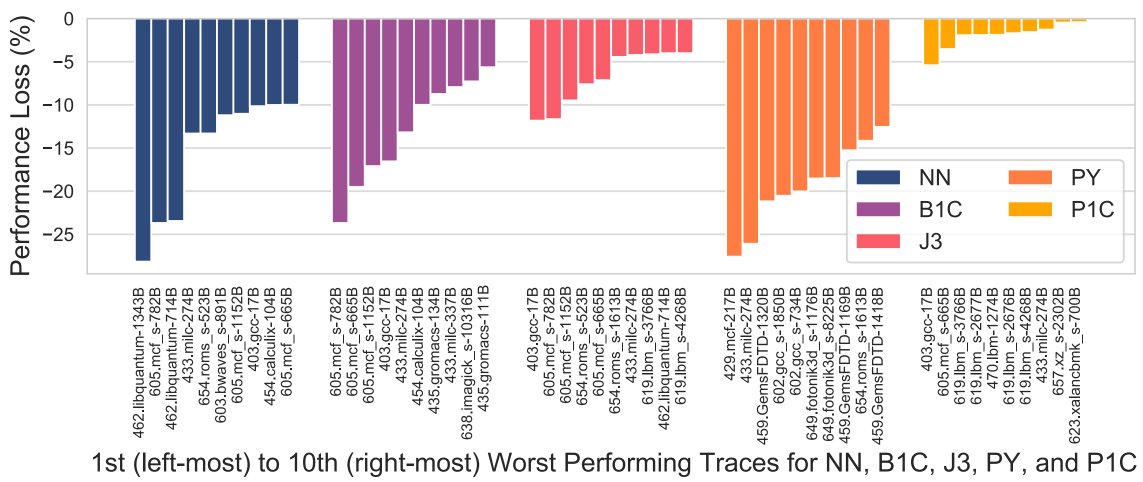


Figure 3·6: Bottom Ten Performance Outliers of 1C - Performance normalized to *SPP*. Each group shows the worst performing traces for NN, B1C, J3, and PY, P1C ordered 10th-worst (right-most) to 1st-worst (left-most).

non-acceptable. Puppeteer has a worst-case loss of only 5% and only 8 traces in total have lower performance than SPP. This clearly illustrates that Puppeteer provides a win-win situation, whereby we not only see a better average performance gain but also see a reduction in the maximum performance loss and the number of traces that have performance loss. For the other prior works, we observe that P1C provides 4.65%, 5.8%, 12.2%, 15.3%, and 5.1% average IPC gain over IPCP, EIP, NN, PY, and J3, respectively.

We also trained two more flavors for Puppeteer and BT using 4C STS and 4C MTS datasets. P4CS and P4CM achieve 0.5% lower average performance gain than P1C. This is because Puppeteer trained using 1C data is better suited for a 1C processor. However, the performance improvement when using Puppeteer, which is trained with 4C data, is still quite large at 45% (average of P4CS and P4CM) compared to a system with no prefetching. This means that Puppeteer generalizes quite well and has learned the underlying architectural phenomena. Compared to B1C for B4CS and B4CM, we observe 1% and 2% lower average performance, respectively. Furthermore, the performance loss for the worst case outlier increases to 30%. Finally, one thing to note about J3 is that its average IPC gain is 2.1% lower than B1C, yet the negative outliers are less in both quantity and magnitude. This is because J3 makes more conservative changes compared to BT since the algorithm cycles through the PSCs as part of a constantly ongoing trial phase. This means that, in a production capable system, J3 might have been more viable compared to BT. This is because processors have to ensure all applications retain or increase their performance across new processor generations.

Puppeteer in a 4C Processor

Here we discuss the use of Puppeteer trained using 1C, 4C STS and 4C MTS datasets, in a 4C processor. These cases are represented by P1C, P4CS and P4CM, respectively. In Figure 3·7, we show the IPC distribution of Puppeteer, and the various prior works in a 4C system while running 4C STS data suite. We observe that the average IPC gain of P1C

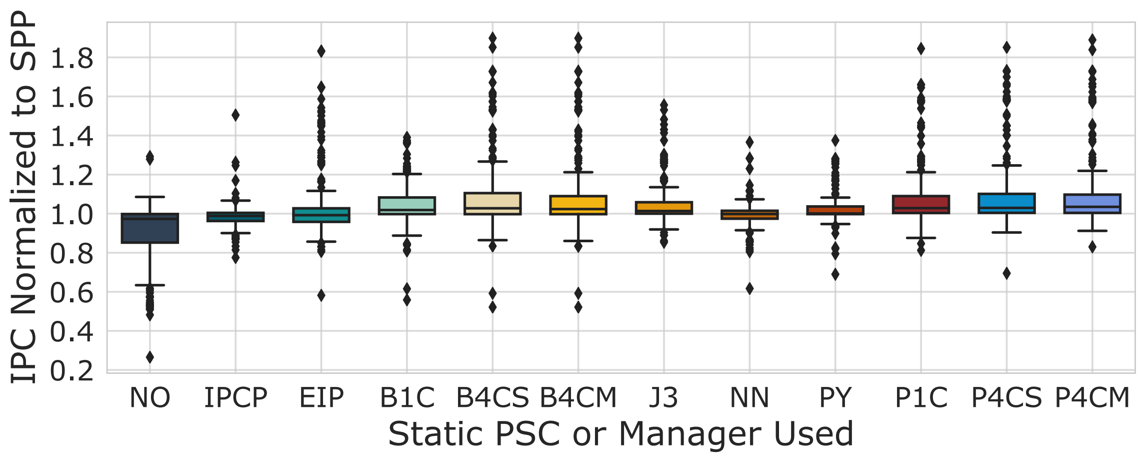


Figure 3·7: Normalized Performance of 4C Processor Running STS - Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all the cores. See Table 3.2 for the notations used along the X-axis.

over the no-prefetcher case is 25.8%. Compared to B1C, IPCP, EIP, NN, PY, and J3, our P1C achieves 4.2%, 10.8%, 4.8%, 8.7%, and 6.8%, 3.7% average IPC gain, respectively. In Figure 3·8, we show the outliers for 4C STS. The number of traces that lose performance is 61 for B1C, 53 for J3, while P1C has just 24 traces that lose performance. J3 has a better worst-case performance compared to Puppeteer because it is taking a much more conservative approach and missing most of the large positive performance gains. The worst-case performance loss in B1C has gone up to 45%, while the worst-case loss of P1C is at only 19%. One key observation here is that although P1C has only been trained on 1C data, it is still applicable to the 4C case and provides a clear advantage over prior work. This is important given that as the number of cores increases, the number of unique combinations of different traces that we will need to run in the multi-core processor increases exponentially (for a 4C processor we need to cover 232^4 = 2.89 billion trace combinations). Therefore, generalization using only 1C data is an important aspect to consider when comparing ML-based algorithms. To further test Puppeteer and BT, we train both algorithms with 4C STS and 4C MTS data suites. For Puppeteer we observe that P4CS has 1.2% and P4CM has

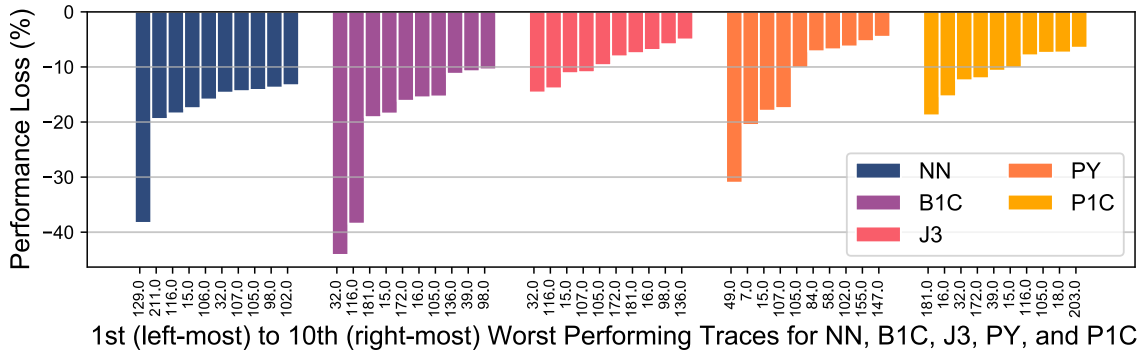


Figure 3·8: Bottom Ten Performance Outliers of 4C STS - Performance normalized to *SPP*. Each group shows the worst performing traces for NN, B1C, J3, and PY, P1C ordered 10th-worst (right-most) to 1st-worst (left-most).

2.2% better performance compared to P1C (see Figure 3·7). While for BT, B4CS has 4.0% and B4CM has 2.9% better performance compared to B1C.

This means that Puppeteer is better at learning the underlying (micro)architectural behavior with a variety of traces running concurrently compared to the same trace running on all the cores. In contrast BT requires data collected specifically from the same set of experiments to achieve better performance.

In Figure 3·9 we show the IPC distribution of Puppeteer and the various prior work when running 4C MTS. P1C achieves 23% average performance gain over no prefetching. P1C's average performance gain is also very close (<0.6% difference) to the average performance gain of P4CS and P4CM. Once again, we observe that Puppeteer is superior to BT in this regard with B1C achieving 5% lower performance gain than P1C. B1C's performance gain is also 4% lower than B4CS and 4.5% lower than B4CM. In Figure 3·10, we show the negative outliers for 4C MTS. B1C has 23 outliers, J3 has 2, while P1C has 3. This means that for the bottom 10 performance cases in P1C, seven of them are actually positive performance gains. In the worst case outlier in B1C has 7% performance loss, J3 has 2% loss, P1C has only 0.4% loss. With high average performance gain and very few outliers, P1C is showing more reliable performance in the complicated MTS bencham-

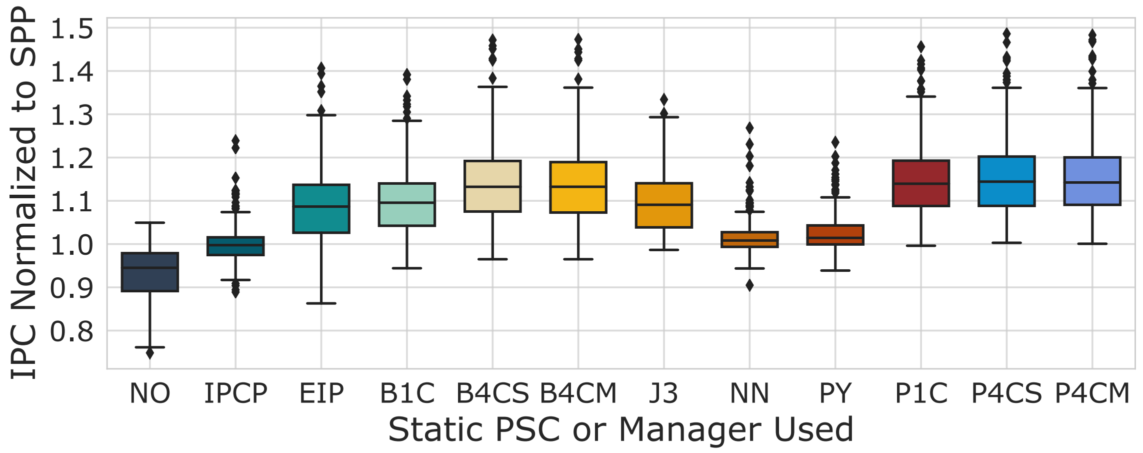


Figure 3·9: Normalized Performance of 4C Processor Running MTS - Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all the cores. See Table 3.2 for the notations used along the X-axis.

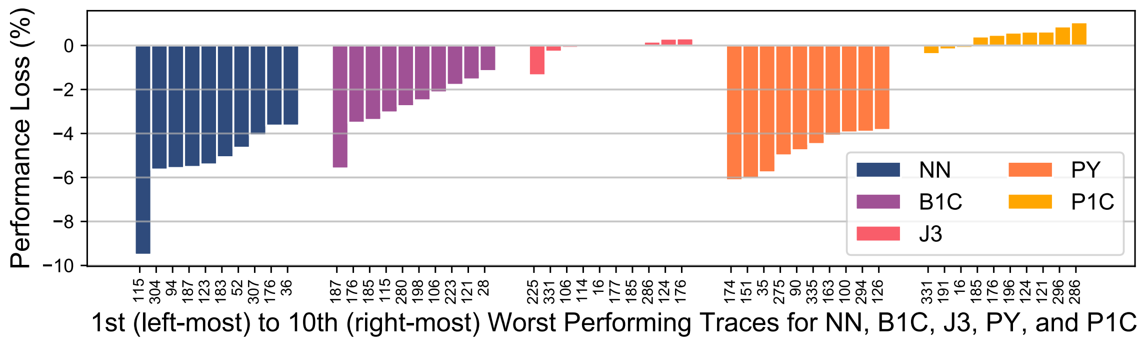


Figure 3·10: Bottom Ten Performance Outliers of 4C MTS - Performance normalized to *SPP*. Each group shows the worst performing traces for NN, B1C, J3, and PY, P1C ordered 10th-worst (right-most) to 1st-worst (left-most).

rks compared to the prior work. Compared to IPCP, EIP, NN, and PY, J3, our P1C (and also P4CS and P4CM) achieves 14.5%, 5%, 12.8%, and 11.9%, 4.4% average IPC gain, respectively.

Puppeteer in an 8C Processor

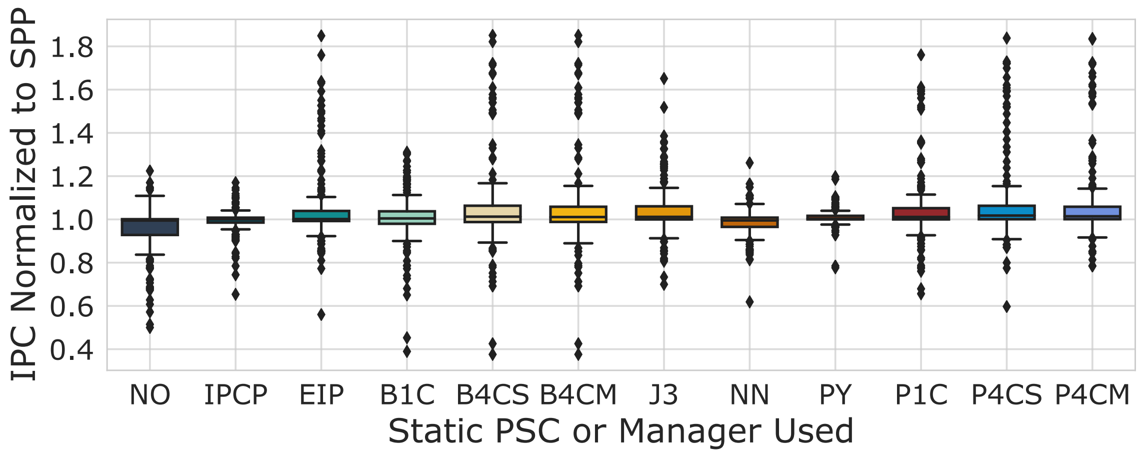


Figure 3·11: Normalized Performance of 8C Processor Running STS - Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all the cores. See Table 3.2 for the notations used along the X-axis.

Here we discuss the use of Puppeteer trained using 1C, 4C STS and 4C MTS datasets, in an 8C processor. These cases are represented by P1C, P4CS and P4CM, respectively. We conduct this 8C processor analysis to test if Puppeteer scales to a larger number of cores. For an 8C processor running STS, we observe several interesting trends (see Figure 3·11). P1C has 11.9% average IPC gain over no prefetching, which is 4.8% higher than B1C. With P4CS and P4CM our IPC gain is 12.7% and 12.9%, respectively, over the no prefetching case. This means that to train Puppeteer for a multicore processor, we should have at least some data corresponding to a multicore processor in our dataset. It should be noted that, compared to the no prefetching case, P1C, P4CS and P4CM have lower IPC gain in the 8C processor than the 4C processor, which in turn has lower performance gain than the 1C

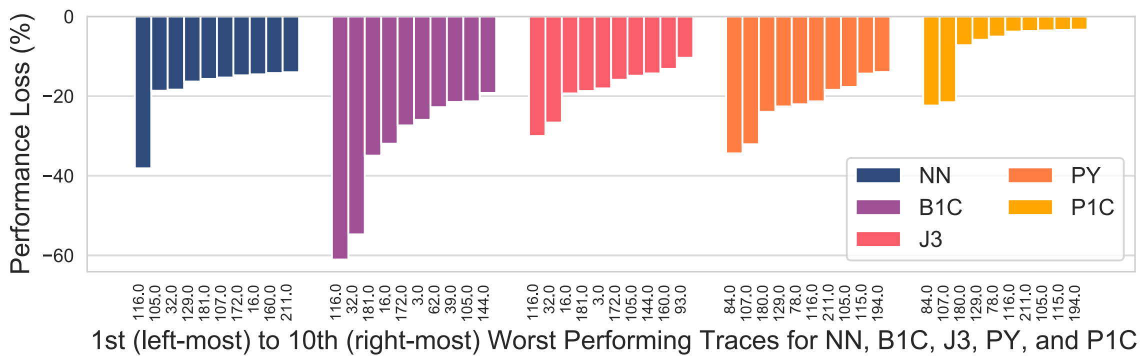


Figure 3·12: Bottom Ten Performance Outliers of 8C STS - Performance normalized to *SPP*. Each group shows the worst performing traces for NN, B1C, J3, and PY, P1C ordered 10th-worst (right-most) to 1st-worst (left-most).

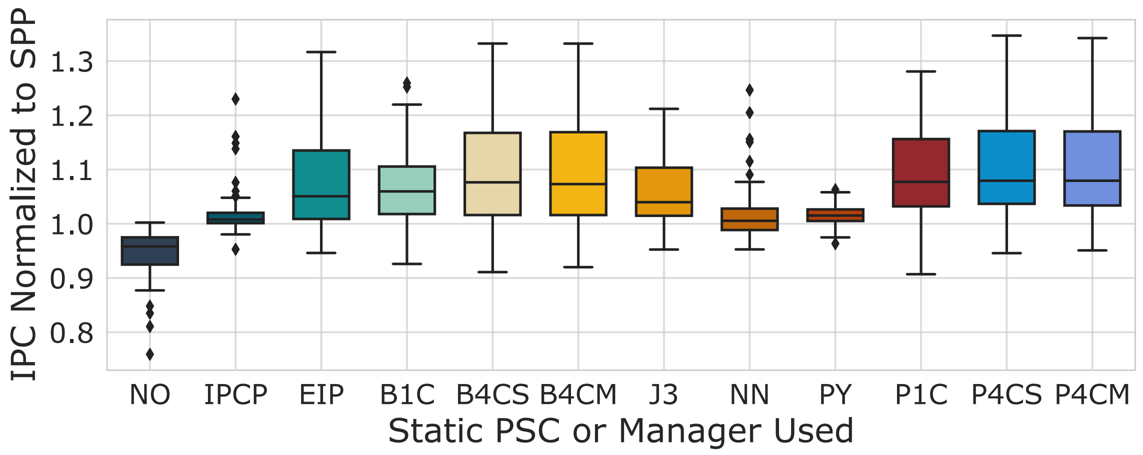


Figure 3·13: Normalized Performance of 8C Processor Running MTS - Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all the cores. See Table 3.2 for the notations used along the X-axis.

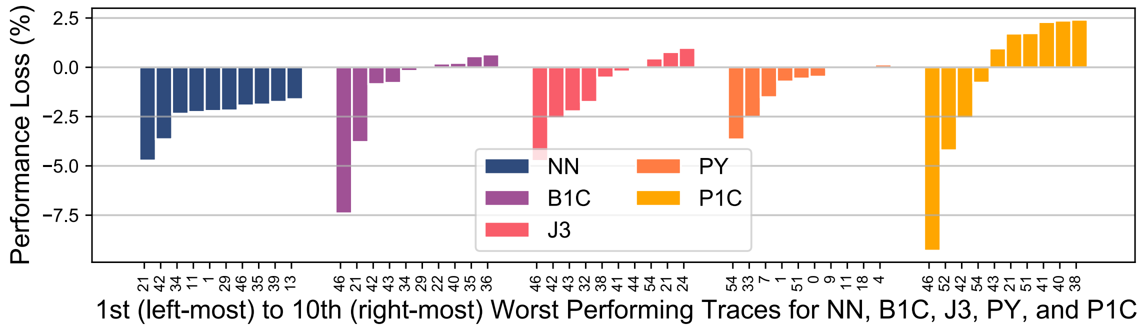


Figure 3·14: Bottom Ten Performance Outliers of 8C MTS - Performance normalized to *SPP*. Each group shows the worst performing traces for NN, B1C, J3, and PY, P1C ordered 10th-worst (right-most) to 1st-worst (left-most).

processor. This is because prefetching is much harder to do in multi-core processors. The overall benefit of prefetching goes down in multi-core processors and Puppeteer has lower possible peak performance. Comparatively, B1C has almost 0% performance gain over SPP. Even a static PSC, namely EIP, has 5.5% higher average IPC compared to B1C. P1C also gives comparable performance to B4CS and B4CM even though P1C was not trained using any data from a multicore processor. Compared to IPCP, EIP, NN, PY, and J3, our P1C achieves 5.9%, 0.2%, 6.5%, and 4.4%, 1.2% average IPC gain, respectively. In Figure 3·12, we show the outliers for 8C STS. B1C has 83 outliers with worst case performance loss of 62%; J3 has 53 outliers with worst case performance loss of 53%; while, P1C has 47 outliers with worst case performance loss of 23%. In the case of P1C, when we get to the 3rd worst case outlier the performance loss drops to only 5%. For the other managers, even the 10th worst case performance loss is larger than 10%.

When using the 8C MTS (see Figure 3·13), we observe similar trends to when using 8C STS. P1C has 2% better average IPC than B1C and comparable performance to B4CS and B4CM. P4CS and P4CM achieve 4% and 4.4% better average IPC compared to B1C. Compared to IPCP, EIP, NN, and PY, J3, our P1C achieves 6.9%, 1.6%, 6.6%, 7.6%, and 3.1% average IPC gain, respectively. In Figure 3·14, we show the outliers for 8C MTS. P1C has a worst case loss of 10% which is the worst loss among all the managers. This is made-up for though with the number of benchmarks that are gaining performance. Even for the bottom 10 performance cases, in the 5th worst performance benchmark we begin to observe 2.5% performance gain. This means that out of all the benchmarks only 5 have less than 2.5% performance gain. In the other managers, this is not the case. These managers are acting much more conservatively, therefore, they have better worst-case performance but they also have very little performance gain.

Puppeteer Performance vs Different Cache Sizes

In this subsection we discuss how the performance of BT, SPP and Puppeteer varies with cache size. We train both models only using data collected on a 1C processor with 32KB L1I\$, 48KB L1D\$, 512KB L2\$, and 2MB LLC. In Figure 3·15 we show the average IPC values of BT (i.e. B1C), SPP, and Puppeteer (i.e. P1C) for 0.5×, 1×, 1.5×, and 2× the nominal L1\$ and L2\$ sizes. P1C is affected by L1\$ size slightly more than L2\$ size but the difference is small (at most 0.4% more performance at 2× L1\$ compared to 2× L2\$). At 0.5× L1\$ size P1C has 1.5% lower performance compared to the performance of P1C at nominal cache size. At 2× L1\$ size P1C has 1.1% better performance compared to the performance of P1C at nominal cache size. At all cache sizes, P1C performs at least 1% better than B1C with the largest performance difference of 2.3% at nominal cache size. SPP performance swings by 5.5% between the 2× L1\$ and the 0.5× L1\$, and by 3.7% between the 2× L2\$ and the 0.5× L1\$. In contrast, P1C has only a 2.9% swing. This means that with P1C we are already operating close to the possible peak performance and so increasing the cache size does not have a significant effect on the performance.

Puppeteer Performance vs Different Hardware Overheads

In this subsection we study the performance of Puppeteer and NN when we have four different hardware overhead budgets – 1KB, 5KB, 10KB, and 100KB. In Table 3.6 we show the various logic and memory components that are required for Puppeteer and NN

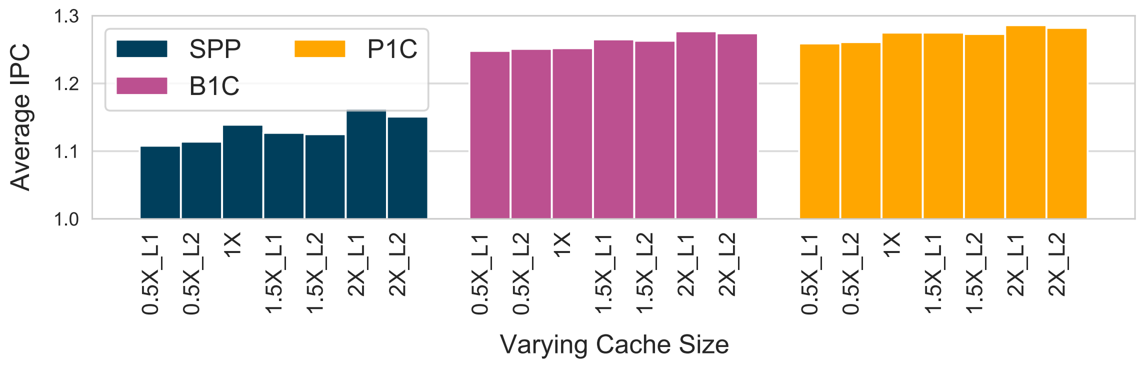


Figure 3·15: Cache Size Sensitivity Study - Average performance of B1C, SPP, and P1C for 0.5×, 1×, 1.5×, and 2× the nominal L1$ and L2$ sizes. For each bar we change only the L1$ size or the L2$ size.

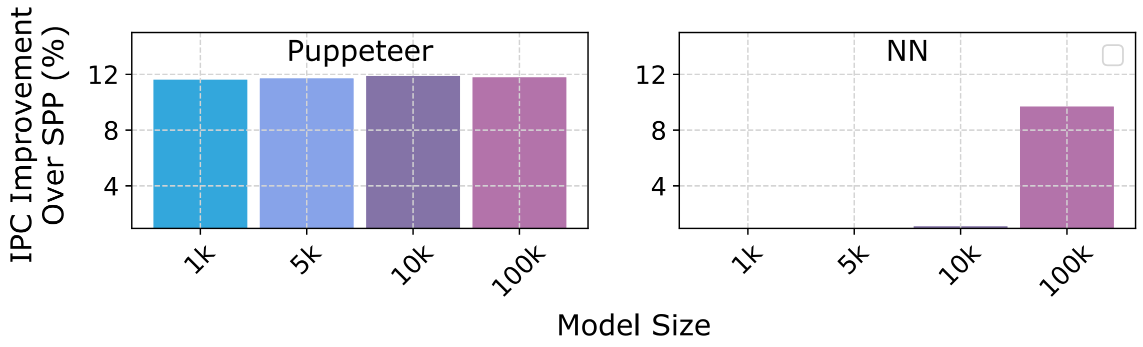


Figure 3·16: Model Size Scaling - Average IPC Improvement on 1C Data Suite of Puppeteer and NN for Different Model Sizes.

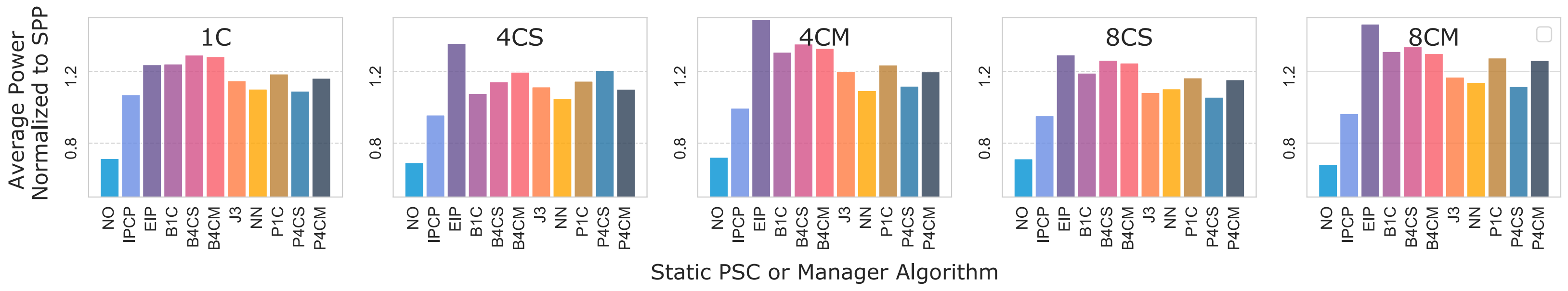


Figure 3·17: Average power of Puppeteer and prior work normalized to *SPP* - Here 1C = 1C data suite, 4CS = 4C single-type trace set data suite, 4CM = 4C mixed-type trace set data suite, 8CS = 8C single-type trace set data suite, and 8CM = 8C mixed-type trace set data suite.

for the different models. We calculate the power, area, and delay values of the components using Cadence Genus and SRAM array compiler for GF22FDX ® (Carter et al., 2016). We show the number of each logic component and SRAM sizes required by each NN. Note that the SRAM word length changes for the differently sized Puppeteer configurations because we require additional bits to address a larger number of nodes. For the NN, the word length is constant since we will traverse all the weights for each computation and do not need addressing (except for a simple register to hold the number of weights in each layer which is negligible)

Here we train a different Puppeteer model and a NN model for each hardware overhead budget. In Table 3.7 we show the total area and power required for the NN and Puppeteer normalized to the 1KB Puppeteer size. All designs in Table 3.7 are such that they need less than 1% of the instruction widow time to choose a PSC.

To select a PSC, Puppeteer just traverses through the random forest for each PSC. If we evaluate all 5 forests in series, where we will require a maximum 250 comparison operations (5 forests $\times$ 5 trees per forest $\times$ (10 comparisons for 1KB, 5KB, and 10KB; 20 comparisons for 100KB) = 250 or 500 comparisons), it will take less than 0.5% of the total time required to execute the 100K instructions in the instruction window (assuming each instruction takes on average a clock cycle). Thus, we end up using the chosen PSC for 99.5% of the instruction window for Puppeteer. In contrast, the NN will require a multiply and addition operation for each weight, as well as a division and ReLu function at the end of each layer to determine the PSC. Due to the heavy computation required, as the size of the NN increases, to choose a PSC in less than 1% of the instruction window time, NN will need to parallelize the operations, which will require additional logic and memory components.

In Figure 3·16 we show the average IPC for 1C for the differently sized models. As it is immediately apparent, Puppeteer provides good performance improvement even when

Table 3.6: Components that Puppeteer and NN use, their area, power, and delay values synthesized using Cadence Genus using GF22FDX® (Carter et al., 2016). We give the number of each logic component required for each NN model size in the bottom table. Puppeteer only requires a single MAX unit for all 4 model sizes. Note that values are normalized due to proprietary reasons.

Logic Components	**Area (µm^2)**	**Power (mW)**	**Delay (Cycles)**
MAX	174	0.28	1
MUL	2358	0.18	1
ADD	559	0.81	1
DIV	10000	11.5	3
RELU	168	0.33	1

SRAM size, Model	**#Bits/ Word, #Words/ SRAM array**	**# SRAM arrays Required**	**Area Norm.**	**Read Energy Norm.**	**Delay (Cycles)**
1KB NN	32, 256	1	1×	1×	1
1KB Puppeteer	38, 256	1	1.3×	1.2×	1
5KB NN	32, 640	2	6.6×	2×	1
5KB Puppeteer	42, 1024	1	2.6×	1.7×	1
10KB NN	32, 864	3	8.6×	1.7×	1
10KB Puppeteer	44, 2048	1	5×	1.9×	1
100KB NN	32, 1024	27	56×	1.4×	1
100KB Puppeteer	50, 16384	1	38×	4.6×	1

# of Logic Component	**1KB NN**	**5KB NN**	**10KB NN**	**100KB NN**
MAX	1	1	1	1
MUL	1	2	3	27
ADD	1	2	3	27
DIV	1	1	1	10
RELU	1	1	1	10

using a small model that fits within 1KB, while NN does not provide any performance improvement until we use a model that requires 10KB. At 10KB model size, a 1C processor with a NN manager has 11.8% lower average IPC as compared to a 1C-processor with Puppeteer. The NN also has $\sim 3\times$ larger area and $11\times$ larger power as compared to Puppeteer. If we compare 100KB NN to 1KB Puppeteer, then using Puppeteer provides 11.6% performance improvement, while NN provides 9.7% performance improvement while the area and power of 100KB NN is $76\times$ and $185\times$, respectively, that of Puppeteer. Therefore, NN is not suitable for hardware prefetcher adaptation.

Table 3.7: Power and Area for various sizes of Puppeteer and NN - Here we use SRAM compiler for designing Node MEM, and design the compute logic using RTL and then synthesize it using Cadence Genus for GF22FDX® (Carter et al., 2016). Power is given over one instruction window. *Norm.* values are w.r.t. to Puppeteer 1KB values.

Algorithm	**Area Norm.**	**Power Norm.**
Puppeteer 1KB	1×	1×
Puppeteer 5KB	2×	1.7×
Puppeteer 10KB	3.6×	1.9×
Puppeteer 100KB	28×	8.3×
NN 1KB	3.4×	3×
NN 5KB	8×	13.8×
NN 10KB	11×	21×
NN 100KB	76×	185×

Puppeteer Power Analysis

In Figure 3·17 we show the average power consumed in the caches for various static PSCs and the prior managers compared to Puppeteer. Compared to the no prefetching case, overall Puppeteer has 65% higher power consumption. Compared to BT (average of B1C, B4CS, and B4CM), Puppeteer (average of P1C, P4CS, and P4CM) has 9% lower average power. While NN and J3 have 6.3% and 2.2% lower average power than Puppeteer, as we have discussed they also have significantly lower average IPC. For the static PSC, EIP is extremely power hungry, with the highest average power that is 20.4% more than Puppeteer. IPCP and NO have lower power consumption than Puppeteer but they also have the worst performance among all options we have considered.

Puppeteer's Impact on Accuracy and Scope

In Figure 3·18 we compare the prefetching scope of the prior works and Puppeteer. Here we determine scope as the number of total misses reduced by the prefetcher or prefetching system, divided by the total number of misses with prefetching disabled. We observe very limited scope across the 4 levels of the memory hierarchy for NN and J3. This is probably one of the reasons these options have lower performance than BT and Puppeteer. Comparing BT (average of B1C, B4CS, and B4CM), Puppeteer (average of P1C, P4CS, and P4CM), in L1D$ and L2$ BT achieves a marginal difference with 0.5% broader scope than Puppeteer. In L1I$ cache BT actually has 3.3% broader scope compared to Puppeteer. However, Puppeteer has 4.8% broader scope compared to BT in LLC. Intuitively, since LLC penalties are more important than L1I$, L1D$, and L2$ penalties, this is a better outcome. Experimentally, our results also support this outcome since Puppeteer has better performance than BT.

In Figure 3·19, we compare the prefetching accuracy of Puppeteer and the prior works. Here accuracy of a prefetcher is measured as the number of misses that a prefetcher or prefetcher system has reduced compared to the case when prefetching is disabled, divided by the number of misses caused by the prefetcher. Puppeteer is better than the other options in L1I$ and L1D$, but comparable in L2$. In LLC, J3 fairs better than the other options with only 1.5% lower accuracy than Puppeteer. In case of BT and NN, Puppeteer achieves 21% better accuracy compared to BT and 28% better accuracy compared to NN. It is also of note that since the accuracy of Puppeteer in L1I$ is 6.7% better compared to BT, this better accuracy plays a role in compensating for the broader L1I$ scope of BT compared to Puppeteer.

Puppeteer-based Temporal variations in PSC

In Figure 3·20 we show the temporal behavior of P1C, B1C, and J3 while running 429.mcf-217B as an example. Figure 3·20a shows the percentage of time for which each PSC was used by each manager algorithm when executing 1.2B instructions of 429.mcf-217B. In Figure 3·20b we show the IPC values and the PSC used in each instruction window for a small slice of the same trace. We would like to note three interesting observations: (i) From Figure 3·20a, both B1C and P1C use *fnl-bingo-spp-no* for the 80% of the instruction windows, yet the performance difference between the two is around 20% over the whole trace. This means the PSC chosen in the remaining 20% of the instruction windows have a larger influence on the overall performance. (ii) In the first 25 instruction windows after

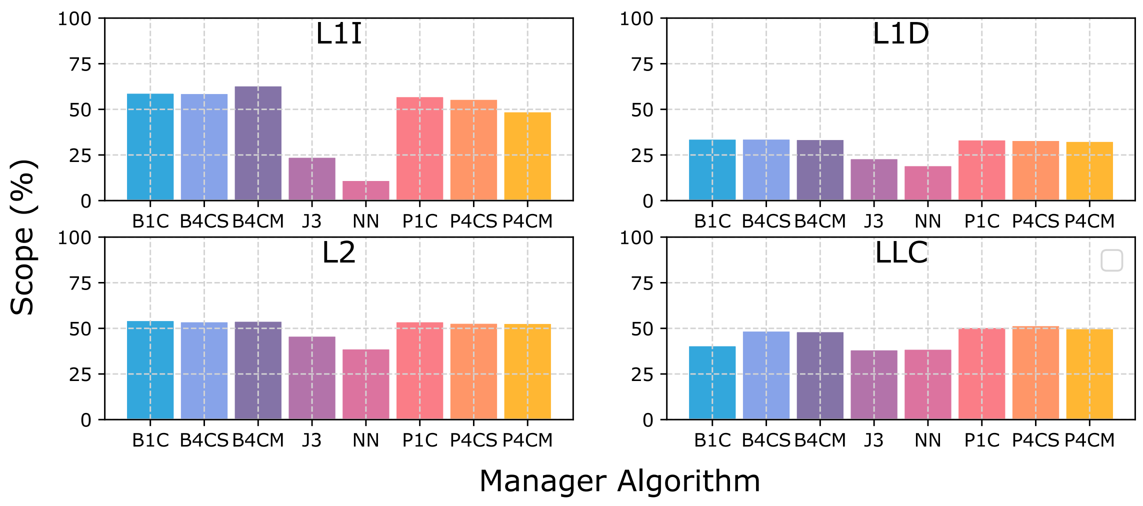


Figure 3·18: Average Scope of Puppeteer and the Prior Works in a 1C Processor - See Table 3.2 for the notations used along the X-axis.

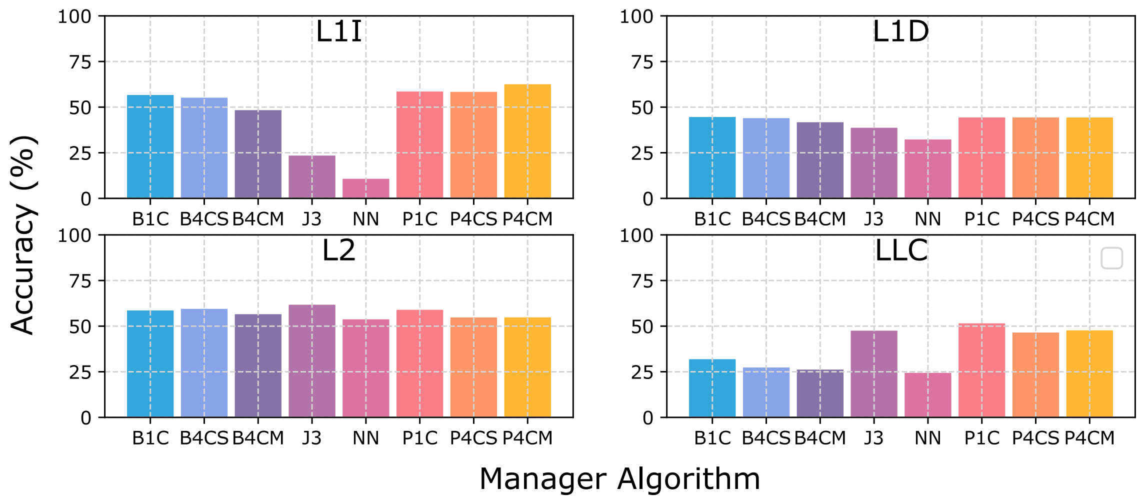
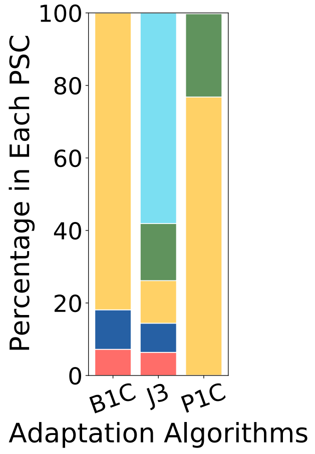
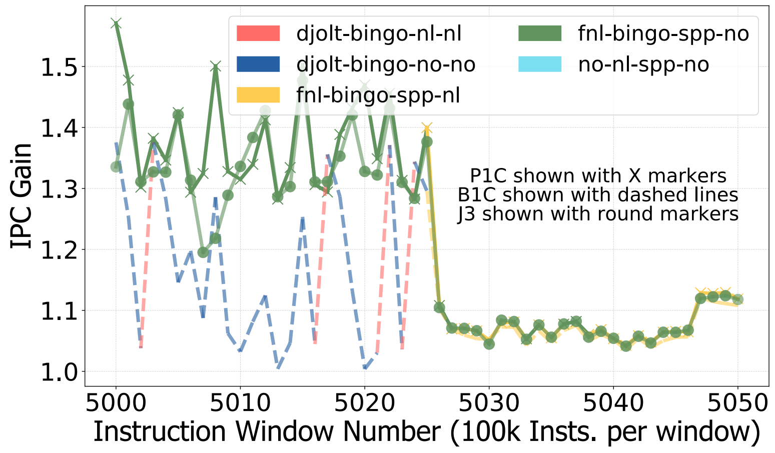


Figure 3·19: Average Prefetching Accuracy of Puppeteer and the Prior Works in a 1C Processor - See Table 3.2 for the notations used along the X-axis.



(a) PSC Percentage



(b) Runtime Behavior

Figure 3·20: Temporal Behavior of Puppeteer- (a) Percentage usage of each PSC for the P1C, B1C, and J3; and (b) IPC gain when using P1C, B1C and J3 across 50 instruction windows and running 429.mcf-217B. For each plot line, for an instruction window we use color coding to indicate the PSC choice. The PSC descriptions are provided in Table 3.5.

the 5000th instruction window, there is a large variation in IPC gain when using B1C as compared to P1C, while all three algorithms converge to the same performance and same PSC during the last 25 instruction windows. This shows that P1C does a better job at predicting the PSC in different regions of an application. (iii) J3 uses the same PSC as P1C yet has lower performance in the first 25 instruction windows. This illustrates that changing the PSC has a cumulative effect on IPC. The choice of PSC made by P1C in prior instruction windows allowed P1C to gain more performance in the given instruction windows compared to J3.

3.4 Summary

In this chapter, we have presented Puppeteer, a novel ML-based prefetcher manager designed using custom tailored random forests. We train a dedicated random forest for each PSC, which allows the random forest to retain more information in a smaller amount of hardware. For the 232 traces that we evaluated, Puppeteer achieves an average performance gain of 46.0% in 1C, 25.8% in 4C, and 11.9% in 8C compared to a system with no prefetching. Puppeteer also reduces the number of negative outliers by 89%. As future work, we will explore a unified design of a ML-based manager that selects from an array of ML-based prefetchers. In addition, we will also consider online training to further improve the performance of Puppeteer.

Chapter 4

SecRLCAD: Securing Standard Cell Libraries Using a RL-based CAD Flow

4.1 Introduction

The immense investment required to continue the state-of-the-art fabrication nearing 20 Billion dollars (Bloomberg, 2020; Inquirer, 2017) has consolidated the number of semiconductors companies doing fabrication or design. On the design side, in only 2021, Nvidia tried to buy ARM (AMD, 2021b), and AMD bought Xilinx (AMD, 2021a), further consolidating the design stack. On the fabrication side, GF recently announced that they are sold up until 2023 and will no longer produce bleeding-edge tech nodes (GF, 2021). The number of fabrication companies capable of the state-of-the-art <10nm nodes is down effectively to three (Alpha, 2021). Unfortunately, the complex requirements of customers coupled with the consolidation of the supply chain have had negative security, supply, and performance impacts on the semiconductor industry (CNBC, 2021a; TechHQ, 2021). The highly fragmented nature of the supply chain for fabrication of IC gives an ample chance for an attacker with knowledge of these companies to insert HT (Zhou et al., 2020).

HTs are malicious third-party circuitry inserted into ICs to compromise the IC chip. There are different types of attacks that HTs can perform. HTs can be utilized as side channels to leak private information from the IC which is a huge issue especially in governmental facilities where HT attacks are increasing (Post, 2018). Destructive HT can damage or even destroy the IC thereby causing performance issues. This can be done by

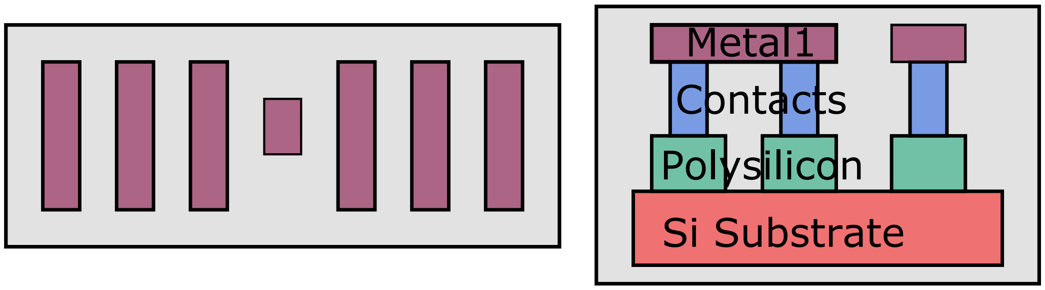


Figure 4·1: Nanoantenna Structure

modifying the layout to reduce PPA, lengthen shortest paths, and insert short-circuits to fry circuitry. Given that IC chips are used in almost every sector of life, from computers, fridges, sensors, automobiles, planes, etc., we must ensure that the security of these chips has not been compromised.

A common insertion technique for hardware trojans is during the fabrication phase (Yang et al., 2016; Rostami et al., 2014; Xue et al., 2020; Dong et al., 2020). Insertion can be done by inserting IP, modifying the design, or changing the layout. Some functional checks can be done to secure against the first two, but it is difficult to functionally check for modifications in the layout. This is especially difficult since attackers can insert tiny HT made from a couple of gates and a capacitor without any functional changes (Yang et al., 2016). Randomly testing for the activation sequence is also near impossible (Yang et al., 2016). Therefore, the only way to check for layout modifications is to physically verify if there have been modifications or not. Such techniques include delayering the chip and SEM imaging to check for HTs. These methods are cumbersome, slow, and expensive. Hence techniques were developed to insert nanoantennas into ICs to generate an optical signature (Zhou et al., 2018; Zhou et al., 2020; Zaraee et al., 2020; Zhou et al., 2015).

We show an example of a nanoantenna in Figure 4·1. It consists of a plasmonic structure in the center and gratings on the two sides. The periodic structures in the nanoantenna cause optical waves to resonate at a particular wavelength and angle of reflectance. With this property, the nanoantenna reflects the optical signal with a powerful response at a certain wavelength, illuminating the structure much more than its surroundings and other wavelengths. We can use backside imaging in Near-IR light to generate a backside image of the IC. We can then compare this image to a known golden design from the IC and a simulated optical response from FDTD simulations. Modifications to the layout will show up as shifting or changing optical signatures giving a reliable and efficient method of checking for HT insertion. The prior art proposes two different methods to insert nanoantennas into standard cells. Zhou et al. (Zhou et al., 2020) utilize high reflectance fill cells to generate optical signatures. This approach does not consider that functional cells could also be modified or changed to generate space for a HT. Zaraee et al. (Zaraee et al., 2020) utilize dual-gate pairs to insert nanoantennas into the IC. This approach requires specific gates to be placed next to each other in the IC. Therefore, current CAD flows would not support the direct insertion of these dual-gate pairs. Even if the CAD flows were modified heavily to force gates next to each other, this would come at the cost of a considerable area overhead in the IC. The current CAD algorithms would not be optimized for such a constraint.

We require a way to directly secure each standard cell from attackers. To accomplish this goal, we propose to insert a nanoantenna into each standard cell directly. The issue here is the manual insertion of a nanoantenna into a standard cell is a design-heavy task. With fill cells or dual-gate pairs, there is ample space to easily insert the grating structures. This is not the case for individual functional standard cells, especially in highly optimized industrial standard cell libraries where the areas of the standard cells are kept at a minimum to save on the total area of the IC. To verify this, we designed and inserted nanoantennas into seven standard cells in GF22FDX (GlobalFoundries, 2006). The design process for a single gate took around 8.5 days on average per gate to get a DRC clean and relatively low area overhead design. In total, for seven min-sized gates, it took two months to proceed onto the synthesis stage. This overhead is unacceptable for securing a modern standard cell library with thousands of gates (GlobalFoundries, 2006).

RL has achieved impressive results at the super-human level in many areas such as GO (Silver et al., 2017), Robotics (Levine et al., 2016; Kober et al., 2013), traffic-light control (Arel et al., 2010), web-traffic routing (Bu et al., 2009), optimizing chemical reactions (Zhou et al., 2017), and more. RL promises good results for anything that can be structured into a *game-like* problem. Following this trend, Ren et al. (Ren and Fojtik, 2021) have shown that by using an RL-based algorithmic flow, they were able to generate DRC-clean standard cells from scratch at or beyond the human-designer level for 92% of the gates.

In this work, we propose to use RL as part of a CAD flow to secure standard cell libraries. Our flow, dubbed SecRLCAD, takes existing standard cells and leverages an RL-based and a graph-cut algorithm to insert the required nanoantenna patterns into the standard cell at a fraction of the time a human designer would take. SecRLCAD works in several steps. First, our graph-cut algorithm searches for an appropriate location to insert a nanoantenna structure by comparing the spacings between the existing vertical metals in a layout. After the initial insertion, the layout is passed to the RL-agent stage, which will iteratively play a game to fix the DRC issues caused during the insertion stage. SecRLCAD is completely end-to-end. SecRLCAD takes a standard cell library as an input, performs the insertion per gate, and outputs all the files of a standard cell library, including the .lef, .gds, and .lib. Therefore, it is easy to directly use SecRLCAD to secure a standard cell library and then use the newly secured library in IC designs with the existing CAD tools.

Our main contributions are as follows:

- SecRLCAD is an end-to-end automatic insertion flow that is modular. SecRLCAD can be easily modified to take a new nanoantenna design, tech node, or additional standard cell designs.
- SecRLCAD shortens the design-time speed from several months to a few minutes compared to a human designer. SecRLCAD can insert the nanoantennas into many of

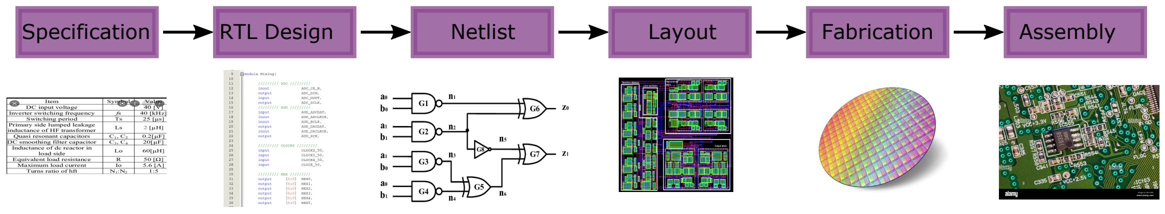


Figure 4·2: Steps for IC Fabrication - In order the chip design process is made up of (1) specification, (2) RTL design, (3) netlist, (4) layout, (5) fabrication, and (6) assembly.

the gates in a few seconds compared to multiple days required by a human designer.

- Unlike the prior work, we secure all the functional cells of a standard cell library, thereby providing complete coverage.

4.2 SecRLCAD: Design

4.2.1 Threat Model

In Figure 4·2, we show the steps for IC fabrication. For our attack model, we assume the insertion of the HT can be done during the fabrication stage. The attacker can modify, shift or replace any standard cell, including fill cells or functional cells, to accommodate the malicious HT blocks. The attacker can get access to the GDSII files used for fabrication. They can generate and insert HT into the IP blocks the victim is using.

4.2.2 Backside Imaging

In Figure 4·3, we show how the nanoantennas create an optical signature for an IC. In near-IR light, the IC's metal regions are reflective while the silicon regions are near transparent. This property is used to design structures that will resonate and reflect the light at certain wavelengths and angles of the light waves. As the designer, we have intimate knowledge of what to expect from the illumination profile for different wavelengths and angles. Therefore, we can secure the IC against modifications for high and low optical response regions using this golden reference. Using backside optical imaging we can extract the full stan-

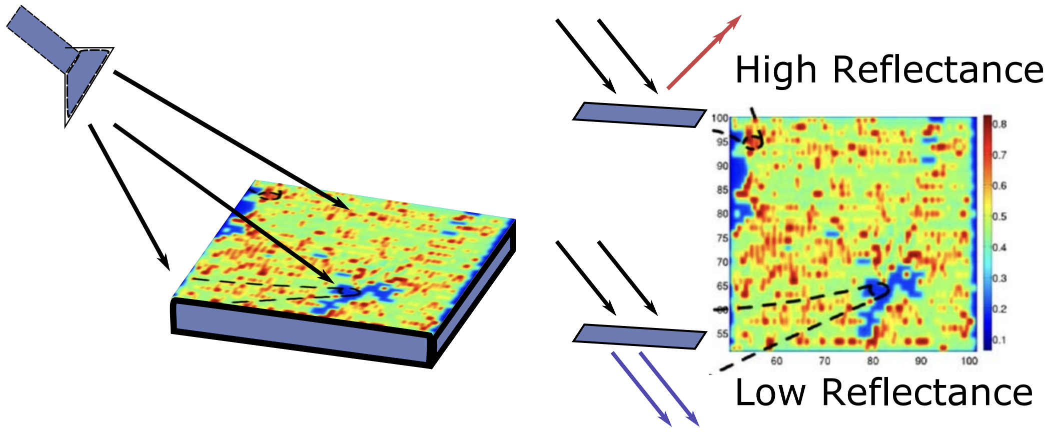


Figure 4·3: Backside Imaging of a Nanoantenna-Inserted IC - (left) The near-IR light illuminates the backside of the IC. (right) Cells that have nanoantennas inserted have high reflectance, the ones that do not have low reflectance. Figure adapted from (Zhou et al., 2020).

dard cell layout. Any modifications made upon the standard cell layouts or movement of the cells will result in a change in the image with high fidelity. Furthermore, we can image a large number of the standard cells simultaneously. Leveraging this high-fidelity and scalable approach provides us a simple, rapid test to check for the tampering of the IC at the fabrication stage.

4.2.3 Single-gate Insertion Compared to Prior Work

We compare the prior two approaches, dual-gate pair layout insertion (Zaraee et al., 2020) and fill cell insertion (Zhou et al., 2020), to our current approach and why our method makes more sense.

Nanoantennas Inserted into a Dual-gate Pair

In Figure 4·4, we provide an example of a dual-gate pair and single-gate layout with nanoantennas inserted. The area is in two gates is naturally larger than a single gate, therefore, the dual-gate pair initially seems like a better idea. Consider in Figure 4·4, with a single gate, we are adding additional metal structures, while in the dual-gate pair we do

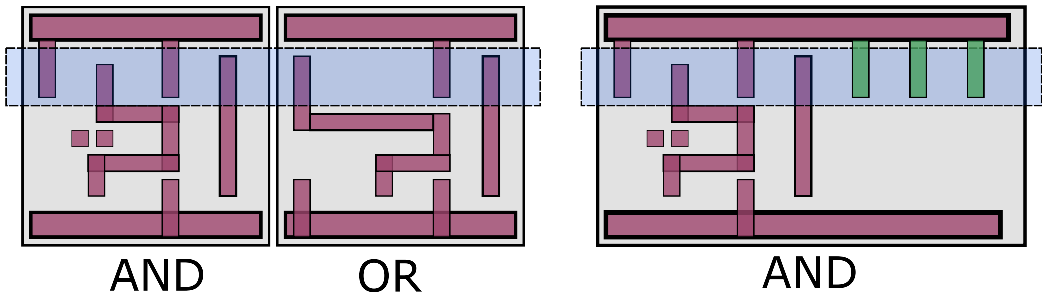


Figure 4·4: Comparison of Dual-Gate Pair Insertion (left) and Single-Gate Insertion (Right) - Newly inserted metals shown in green. Blue rectangle marks the location of the nanoantenna structure.

not need to add any structures since we already have the seven required vertical metals. Therefore, it initially seems like we have 0% overhead for two gates, and 60-70% overhead for a single gate. The issue here is the dual-gate pairs have to be placed next to each other for this to work. Fundamentally, while there might be instances where the gates we want to be placed next to each other naturally get placed next to each other, for most of the cases, without any constraint on the placement algorithm, this will not be possible. Therefore, the overhead will be closer to having a redundant logic gate next to every gate. If we want 100% coverage, the overhead will be closer to 100% in dual-gate pairs.

Nanoantennas Inserted into a Fill Cells

In Figure 4·5, we show the illumination profile of an IC with nanoantenna-inserted fill cells. Zhou et al. state that if a fill cell is replaced or shifted, the illumination profile will change hence the HT can be detected. The problem here is that fundamentally we wish to reduce the number of fill cells when designing IC. In modern IC, the fill cell to total area percentage is less than 20% (Siemens, 2022b). As shown in Figure 4·5, the fill cell percentage is less than 20% of the whole IC. Such a low coverage percentage gives ample functional cells for the attackers to modify to insert HTs.

With SecRLCAD, both of these issues no longer exist. SecRLCAD modifies the layout of each one of the functional standard cells and inserts a unique nanoantenna design to

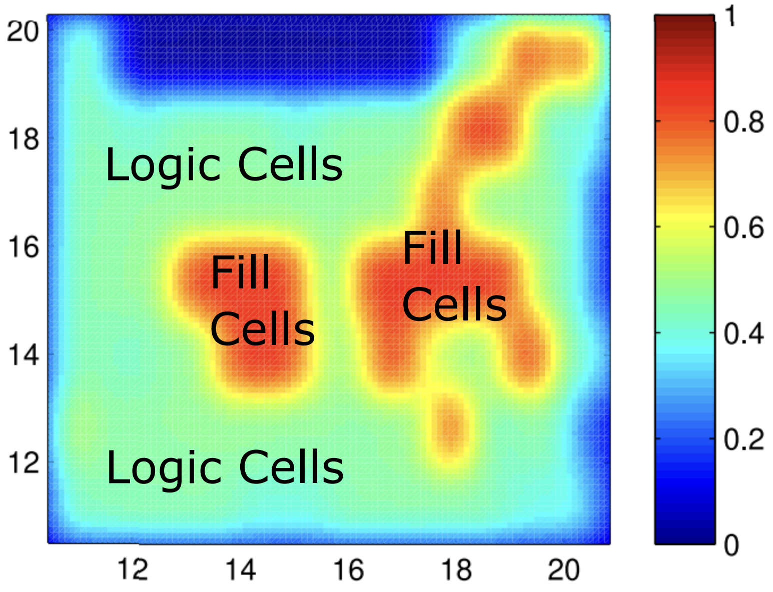


Figure 4·5: Illumination Profile Using Only Fill Cells - High-reflectance areas are where the fill cells reside. Figure adapted from (Zhou et al., 2020).

achieve a secure standard cell library. Our flow secures each gate and is compatible with the existing CAD flows, thereby, it is superior to the dual-gate pair method. Additionally, since SecRLCAD secures the functional standard cells, we can use these functional cells to achieve 100% coverage in the non-fill cells.

4.2.4 Manual Insertion Steps of a Nanoantenna into a Single Gate

To test our idea, we manually inserted nanoantennas into seven minimum-sized standard cells designed by Invecas from the GF22FDX (GlobalFoundries, 2006) technolgy node. While fairly simple to explain, the process is challenging to go through due to the intense design labor that is required.

Based on human intuition and how much the current design resembles the nanoantenna layout, we find a logical location to insert the nanoantenna and shift the metals around to create room to insert the nanoantenna. Next, we check for DRC problems and shift metals around to fix the DRC problems. The main reason for design-heavy work is fixing the

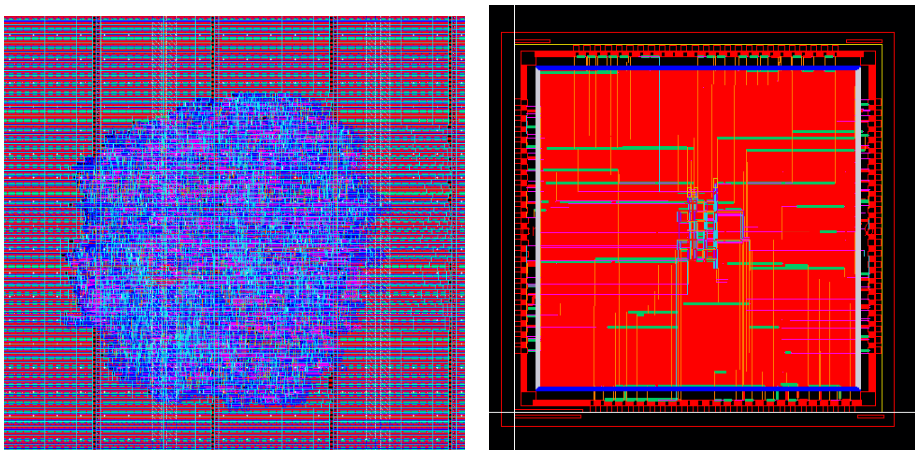

(a) Zoomed in On a FIR Filter (b) Overview of IC Chip

Figure 4·6: Images of Our Tapeout Done in GF22FDX - (left) shows a zoomed in region with an IC of interest. (right) shows the overall IC chip with IO.

DRC issues. Many DRC issues that do not even appear at the standard cell level pop up when inserted into a larger design. This requires an iterative process for fixing the errors. Once we have gotten all the standard cells DRC clean, we generate a .lef file from abstract generator and a .lib file from liberate. Using the new standard cell library and files, we push several IC designs through the ASIC flow to get the GDS. End-to-end, the whole process took four months of extremely grueling design, with almost two months dedicated to initially designing the standard cells. Figure 4·6 shows the final IC using our secure standard cells.

The problem with this approach is the design-time overhead. Even with two designers, the whole process was tediously long. We can encode the same "search for a similar location to insert the nanoantenna, then fix the DRC issues that appear" logic into an algorithmic form. Having an algorithmic approach will save us a lot of time, given that standard cell libraries are typically made out of several hundreds to thousands of different cells.

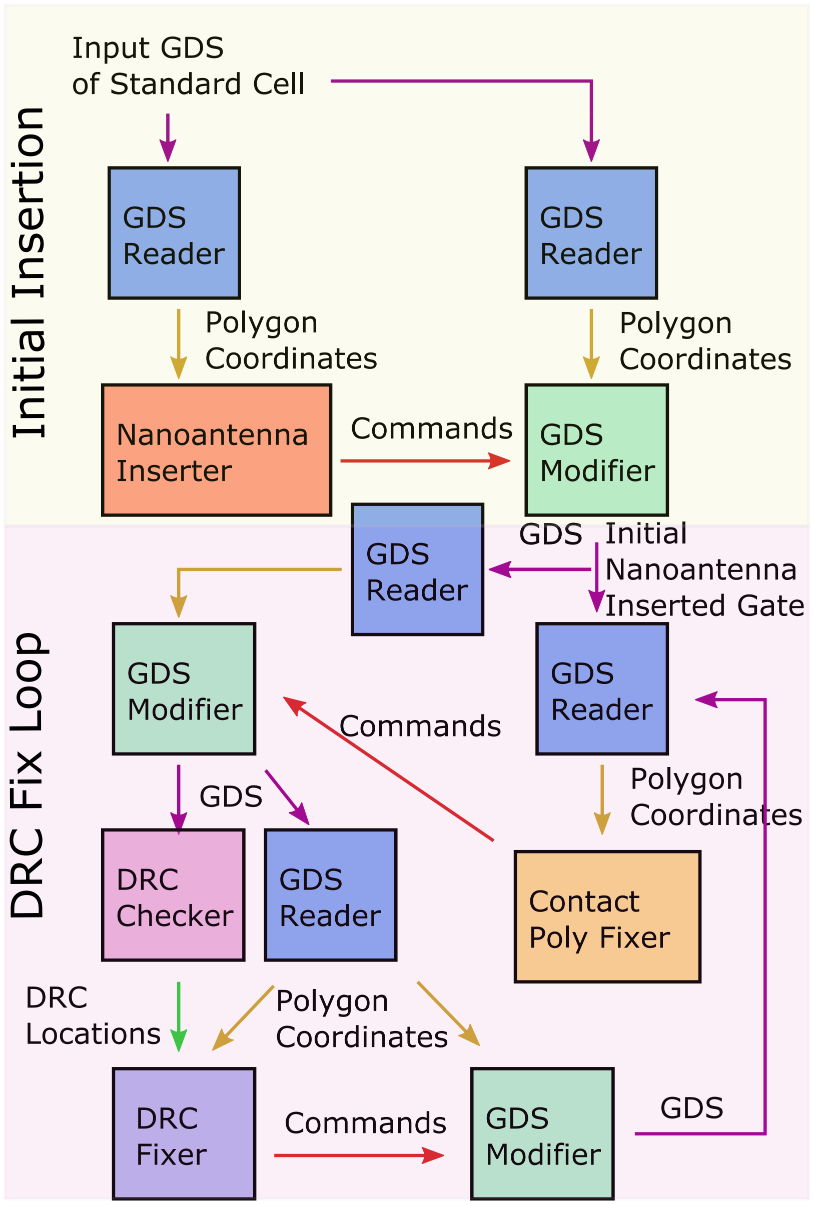


Figure 4·7: Nanoantenna Insertion and DRC Fix Flow.

Table 4.1: SecRLCAD Notations.

Notation	Notation Meaning
N	Number of metal polygons in the initial pre-insertion standard cell layout.
K	Number of metal polygons inserted for the nanoantenna structure.
m	Number of metal polygons we can use from the initial standard cell layout for the nanoantenna structure.
x_{ij}	X-axis coordinates of a given metal polygon i.
y_{ij}	Y-axis coordinates of a given metal polygon i.
y_{cut}	Y-axis location that we select to insert the nanoantenna.
ε	Mathematical value for infinitesimal value.
H_i	Height of metal polygon i.
W_i	Width of metal polygon i.
H_g	Height of grating structure.
W_g	Width of grating structure.
H_a	Height of center plasmonic structure.
W_a	Width of center plasmonic structure.
L_{m1}	Minimum distance between two metal polygons as defined by the technology node.
$c_1, c_2, and c_3$	Scaling coefficients to balance the three terms in the optimization function.
$\Delta area$	Difference in area.
Δx_{ij}	Difference in x_{ij}.
Δy_{ij}	Difference in y_{ij}
$s(t)$	State at a given time t.
$a(t)$	Action taken at a given time t.
$Q(s,a)$	Q-table entry for a given state and action pair.
$Q(s',a')$	Q-table entry for the next s' and a' pair.
γ	Discount value for the learning function.
α	Learning rate for the learning function.
r	Reward value for a given action state pair.
θ	Weights of the DQN.
θ'	Future weights of the DQN.

4.2.5 SecRLCAD Flow: Overview

We now explain our main contribution, designing an algorithmic flow capable of re-designing an existing standard cell library to be secure. The logic is very similar to the human-intuition-based approach. Broadly, SecRLCAD is made of two portions. The first is to perform the initial insertion. The second is to fix any DRC issues that pop-up. While performing these two tasks, SecRLCAD tries to make the area smaller.

In Figure 4·7, we show the primary flow of SecRLCAD and the blocks that insert the nanoantenna, then fix DRC issues. In Table 4.1 we list the notations we use to describe SecRLCAD.

We describe SecRLCAD at a high level as:

1) SecRLCAD begins with the existing standard cell library and starts from a single

Algorithm 1 SecRLCAD Insertion and DRC Fix Flow.

```
1) Using Nanoantenna Inserter find $y_{cut}$ to insert the nanoantenna.
2) Using Nanoantenna Inserter modify the layout to insert the nanoantenna.
3) Send commands to GDS Modifier to modify the GDS file.
4) Using Contact Poly Fixer to fix contact and poly issues.
5) Check for DRC issues using DRC Checker and try to fix DRC issues with DRC Fixer.

Repeat step 5 until all DRC issues are fixed.
```

GDS file from this library.

2) Our `GDS Reader` transforms the GDS file into a python readable polygon coordinate format.

3) These coordinates are fed into the `Nanoantenna Inserter` to find the location for the nanoantenna insertion.

4) `Nanoantenna Inserter` issues commands to the `GDS Modifier` to insert the nanoantenna design into the GDS layout. The `GDS Modifier` then uses the commands and changes the GDS file while maintaining logical connections from the original design.

5) `Contact Poly Fixer` shifts the contacts and poly layer around to maintain logical connections with the original metals that were already existing in the standard cell layout. The `Contact Poly Fixer` fixes any DRC issues that occur related to the contact and poly layer by using these shifts.

6) SecRLCAD begins a loop of checking for DRC issues using `DRC Checker`, fixing metal DRC issues with `DRC Fixer`, then fixing contact and poly DRC issues with `Contact Poly Fixer`. This loop is continued until we get a DRC clean design.

In Figure 4·8, we show the process of creating a standard cell library after securing the standard cells. Once we have all gates inside a standard cell library secured, we automatically generate .lib and .lef files using our secondary flow. At the output, SecRLCAD will have created a secure standard cell library.

We will now describe each block of SecRLCAD in more detail.

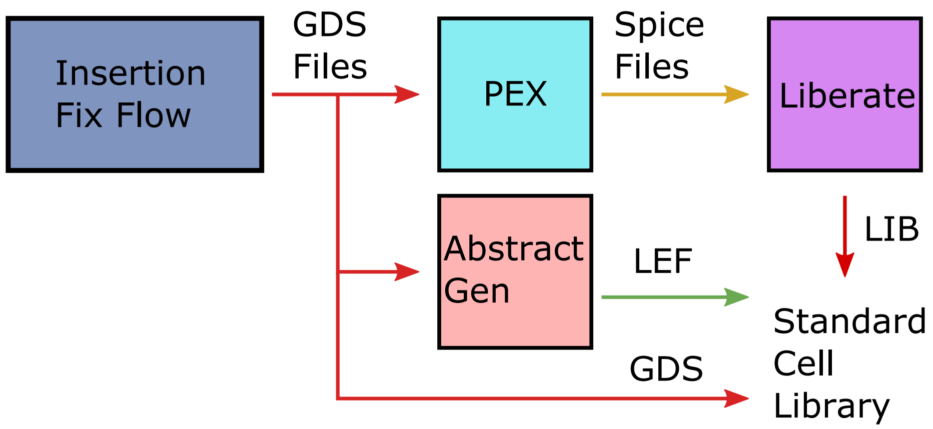


Figure 4·8: Standard Cell Library Creation Flow.

GDS Reader

`GDS Reader` is made up of two stages, a *read-in* stage using Gdspy library (Gdspy, 2022), and a *split-into-blocks* stage to separate polygons into rectangles. For the `DRC Fixer` and `Nanoantenna Inserter` blocks, the manipulation of the metal layer is difficult to accomplish with simple functions upon polygons. Therefore, splitting the metal polygons into rectangles and applying modifications on the rectangles makes the command structure more streamlined and easier to represent.

Nanoantenna Inserter

`Nanoantenna Inserter` is the first of the three algorthmic blocks (with the other two algorithmic blocks being the `Contact Poly Fixer` and the `DRC Fixer`). `Nanoantenna Inserter` is made of several stages. First, we search for the best insertion cut. Next, we shift and modify the polygon shapes to satisfy the nanoantenna design we wish to reach.

In the first stage, given N metal polygons in the layout, the VDD and GND lines can not be used as grating structures. There are in total $N-2$ metal polygons to choose from. If we need to insert K metal gratings and assuming we can use m metal polygons from the $N-2$ original metal polygons, we need to insert $K-m$ metal grating structures. In total we have the following options:

$\binom{N-2}{K}$ + $\binom{N-2}{K-1}$ + $\binom{N-2}{K-2}$ + ... + $\binom{N-2}{0}$ = $\sum_{k=0}^{K}\binom{N-2}{k}$

This combinatorial sum does not have a closed form. If $K > (N-2)/2$ the algorithm runs with $O(2^N)$ i.e., the runtime is exponential. But since $K < (N-2)/2$ in most cases, the brute force solution runs in polynomial time. In order to speed up our run time we come up with a linear time algorithm. In the case that the algorithm can not find a good solution we can return to the initial layout, prior to nanoantenna insertion, and start with another point.

We can represent a polygon by its two corners. Let us denote the x-coordinates of this polygon as $\{x_{ij}\}$ and the y-coordinates of this polygon as $\{y_{ij}\}$, where i = 1,...,N polygons and j = 1,2 for the bottom-left and top-right corners.

We want to pick a y_{cut} where $min(y_{ij}) < y_{cut} < max(y_{ij})$ and $y_{cut} \in \{y_{ij} + \varepsilon\} \cup \{y_{ij} - \varepsilon\}$ for all i, j. In the worst case, we have $2N$ number of cuts to try out which is linear solution space. For a given y_{cut}, we select from the set of possible cuts, $y_{cut} > y_{i1}$ and $y_{cut} < y_{i2}$ and $W_i < H_i + \varepsilon$. Here W_i is the width of a polygon and is defined as $x_{i2} - x_{i1}$ and H_i is the height of a polygon and is defined as $y_{i2} - y_{i1}$.

For each polygon that y_{cut} goes through, we calculate the similarity between the polygon and the required grating shape. We define the similarity as the overlap of the polygon with the grating shape $min((W_i - W_g) * (H_i - H_g))$ where W_g is the required width of the grating structure and H_g is required height of the grating structure.

We also add a second term for available free space around the polygons. The total free space is the total length between the maximum and minimum x_{ij} minus the length required to fit the m nanoantenna grating structures, i.e., $max(x_{ij}) - min(x_{ij}) - m * L_g$ where x_{ij} is a coordinate that is part of a polygon that y_{cut} went through and L_g is the grating period.

Once we have a best y_{cut} that provides the maximum number of polygon cuts, there are three possibilities. The first possibility is when $m > K$. Since we need K polygons, we can choose a subgroup from $m - K + 1$ different possible polygons. We choose the one that maximizes the similarity and free space. The second possibility is when $m = K$. We directly choose these m polygons. The third possibility is when $m < K$. We add metals to the free space between the polygons or on the sides until we reach $m = K$.

After choosing the K polygons we wish to reshape the spaces to match the shape of the nanoantenna. To do so we design the second stage as follows, we want to minimize the maximum horizontal length to decrease the area of the gate. We also wish to minimize the movement of the polygons as much as possible from their original spot in the layout prior to nanoantenna insertion. Finally, we want to minimize the area difference between the grating structure of the nanoantenna and the original polygon.

The optimization algorithm is as follows:

$\min(c_1(\max_{i,j}\{x_{ij} + \Delta x_{ij}\} - \min_{i,j}\{x_{ij} + \Delta x_{ij}\}) + c_2(\sum_{i=1}^{N}(\sum_{j=1}^{2}\Delta x_{ij} + \sum_{j=1}^{2}\Delta y_{ij}))$

$+c_3 \sum_i \Delta area_i)$

Subject to seven constraints:

(i) The original connections of the polygons must be sustained. If a polygon is moved or extended the connecting polygon will also shift or extend accordingly to sustain the connection. Mathematically, given two polygons n and l, if $x_{n1} \leq x_{li} \leq x_{n2}$ and $y_{n1} \leq y_{li} \leq y_{n2}$, this property must be conserved for any two polygons after all actions are taken.

(ii) New connections must not be created. If a polygon is not touching another polygon in the original layout, it shall not touch this polygon with any movement. Mathematically, given two polygons n and l, if $x_{li} > x_{n1}$ and $x_{li} > x_{n2}$ and $y_{li} > y_{n1}$ and $y_{li} > y_{n2}$, this property must be conserved for any two polygons after all actions are taken.

For constraints (i) and (ii), we track the polygon connections with a graph algorithm to sustain connections.

(iii) The minimum distance between metals polygons, as defined by the technology node, will be sustained. Mathematically, given L_{m1} and two metal polygons m and n, we will continue taking actions until $x_{mi} - x_{ni} > L_{m1}$ and $y_{mi} - y_{ni} > L_{m1}$.

(iv) There is a known distance L_g between the grating structures, the final design must have this distance between the grating structures. Mathematically, given two metal polygons n and m that are part of the nanoantenna grating structure, we will continue taking actions until for all n and m pairs, $x_{m1} - x_{n2} > L_g$.

(v) The grating pairs have a known H_g and W_g that they must have in the final design. The algorithm will continue to modify the layout until these shapes are also met. There is $\pm$ 20% flexibility in height and width for this constraint. Mathematically, given a metal n that is part of the grating structure, we will continue taking actions until $1.2 * W_g > W_n > 0.8 * W_g$ and $1.2 * H_g > (H_n) > 0.8 * H_g$.

(vi) The center nanoantenna structure has a known H_a and W_a, the algorithm will continue until the center has this shape. Mathematically, given a metal n that is the center plasmonic structure, we will continue taking actions until $W_n = W_a$ and $H_n = W_g$.

(vii) The distance between the VDD and GND distance must be sustained and can not be changed. Mathematically, $max(y_{ij}) - min(y_{ij})$ will be kept constant.

Based on these constraints, we can move and extend the polygons until we reach the nanoantenna design.

GDS Modifier

`GDS Modifier` has three main functions: (i) *Extend*, (ii) *Move*, and (iii) *Add*. With the *Extend* command SecRLCAD can extend polygons in the positive or negative direction, i.e., contract the shape by a given amount. With *Move* command SecRLCAD moves any polygon to a given direction by a given amount. Finally, with *Add* command SecRLCAD can add in new metal polygons to the layout in order to insert, if required, the extra gratings for the nanoantenna structure.

Contact Poly Fixer

`Contact Poly Fixer` is the simplest of the three algorithmic blocks (with the other two algorithmic blocks being the `Nanoantenna Inserter` and the `DRC Fixer`). Essentially, most poly/contact issues are related to the functions of the `GDS Modifier` are either mismatched overlap issues, i.e., the metal layer has shifted, or are related to the metal width not leaving enough width at the sides of the contacts, i.e., shrinking metal widths. The first issue is easy enough to fix. `Contact Poly Fixer` shifts the contacts by the same amount as the metal that the contact was connected to. For the second issue, we define a minimum metal thickness that the metals with contact connections should not go below. Using this as a hard limit, we preemptively remove this issue from occurring. As discussed in Chapter 5, in the future, with flows that build the standard cells from scratch, this block will have to be more intelligent.

DRC Checker

`DRC Checker` relies on external tools to check for DRC issues in the layout. The external tool returns how many DRC issues exist in the layout. `DRC Checker` parses the DRC report and separates the metal DRC issues and the other issues.

DRC Fixer

The `DRC Fixer` is the third of the three algorithmic blocks and the most complex. When building the `DRC Fixer` block we need an algorithm that is capable of making sequential decisions on an environment. We begin by explaining Q-learning to explain the logic behind the `DRC Fixer` block and then explain Deep-Q-Networks (DQN) which is the underlying algorithm of the `DRC Fixer` block. In general, for Q-learning, we build a memory table $Q(s,a)$ to store Q-values for all possible combinations of state (s) and action (a). For a given environment we can take certain actions upon this environment and we can maximize our reward by taking these actions in a certain order.

The algorithm for Q-learning is:

Algorithm 2 Q-Learning Algorithm

```
1) Initialize table Q(s,a) with random values.
2) Take a action (a) with epsilon — greedy policy and move to next state s'.
3) Update the Q value of a previous state by following the update equation: $Q(s,a) = Q(s,a) + \alpha(r + \gamma max(Q(s',a')) - Q(s,a))$.
4) Repeat M steps.

Here $\alpha$ is the learning rate, r is the reward, and $\gamma$ is the discount.
```

Here, epsilon — greedy means we move between taking a random action (exploration) and taking an action based on Q-value (exploitation) based on the learning rate (α).

For our problem, the `DRC Fixer` must be capable of understanding and understanding complex relational geometric properties in the layouts to figure out which polygon shapes are allowed with relation to its' surrounding polygon shapes. The number of stages that the metal layout could observe is $N > 10^{1250}$ (Ren and Fojtik, 2021). Since this number is larger than the size of the known universe, there has to be a learning methodology for this stage to generalize to the possible cases. DQN comes into the picture when the number of $Q(s,a)$ are too large to memorize.

DeepMind developed DQN (Mnih et al., 2015) in 2015. The algorithm is inspired by the deep networks used to achieve very high accuracy in image recognition, another field where the number of unique states is incomprehensibly high. DQN is successful due to four traits: (i) Experience Replay, (ii) Target Network, (iii) Clipping Rewards, and (iv) Skipping Frames. *Experience Replay* uses state and reward storage to avoid overfitting, which is a huge concern in RL-based methods. *Target Network* fixes the parameters of the target network every several thousand steps to avoid unstable target functions. *Clipping Reward* clips all rewards to between 1 and -1 to make training for different tasks stable. *Skipping Frames* skips over frames of a game. Given humans also do not decide on actions for every single frame of a game, this makes logical sense.

The goal of DQN is to learn to approximate a complex, nonlinear function Q(s, a) instead of memorization using a DNN. We can train the DNN with the following loss function: $Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$.

During the learning process we use two separate Q — networks to calculate the predicted value (weights θ) and target value (weights θ'). The target network is frozen for a duration and then the target network weights are updated by copying the weights from the actual Q network.

We can write the DQN algorithm as follows:

Algorithm 3 DQN Algorithm

```
1) Initialize the replay buffer with random actions and initialize Q(s,a).
2) Select an action (a) using the epsilon - greedy policy.
3) Store action output in the replay buffer as $< s, a, r, s' >$.
4) Sample some random batches of transitions from the replay buffer and calculate the loss.
5) Perform gradient descent with respect to actual network parameters in order to minimize the loss.
6) After every k steps, copy our actual network weights $\theta$ to the target network weights $\theta'$.
7) Repeat for M steps
```

4.3 Evaluation

4.3.1 Methodology

To build SecRLCAD, we rely on several tools and frameworks. `DRC Fixer` uses the OpenAI Gym (Brockman et al., 2016) and stablebaselines3 (Raffin et al., 2021) framework to train the DQN. `DRC Checker` has been built to support KLayout (Klayout, 2022) and Calibre (Siemens, 2022a). `DRC Reader` and `DRC Modifier` rely on the Gdspy library (Gdspy, 2022) to read and modify the GDS files. To generate .lef files we use Cadence Abstract

Table 4.2: FIR Filter Place and Route (PnR) Results

Standard Cell Library	**Area (μm^2)**	**Number of Gates**
Invecas Cells	902.7	3012
Seven Secure Standard Cells	1339.4	4471

Generator (Cadence, 2022b). To generate .lib files we first need to generate .sp files. To accomplish this, we use Calibre PEX (Siemens, 2022c). Once we have the .sp files, we can use Cadence Liberate (Cadence, 2022a) to generate the .lib file.

For the technology node, we experiment with GF22FDX (GlobalFoundries, 2006) for the initial tapeout. To test SecRLCAD more extensively, we use *Nangate FreePDK 45nm* (Knudsen, 2008). We build the rest of our pipeline in python.

4.3.2 Results

Manual Implementation in GF22nm

In Table 4.2 we provide our preliminary analysis done in GF22FDX. In the manually designed standard cell library we have AND2, OR2, INV, XOR2, XNOR2, NAND2, and NOR2 gates. For more complex gates, such as gates with more than two inputs we use more than a single gate. This results in the number of gates going up by $\sim 50\%$ in the FIR filter. The area of the total design has also increased by $\sim 48\%$. A large portion of this area increase can be associated with the above problem of having access to fewer standard cell types with only two inputs since designing using larger gates would mask the overhead of the nanoantennas even better. Despite this, using a dual-gate insertion approach is worse since this would result in a $\sim 100\%$ increase in area.

SecRLCAD Results

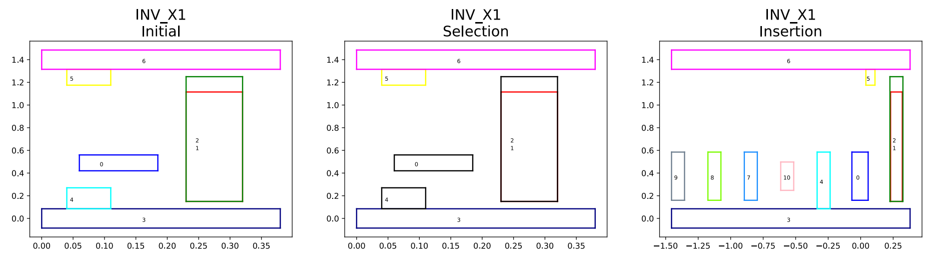
Figure 4·9 and Figure 4·10 show the insertion of the nanoantenna for seven gates: INV_X1, AND2_X1, OR2_X1, NAND2_X1, NOR2_X1, XOR2_X1, and XNOR2_X1. Here we begin with the initial layout, find the polygons to be used as grating structures, then follow the

Table 4.3: Secured Standard Cells in *Nangate FreePDK 45nm* - Part 1 - Percentage increase of the area of each standard cell compared to its non-secure version.

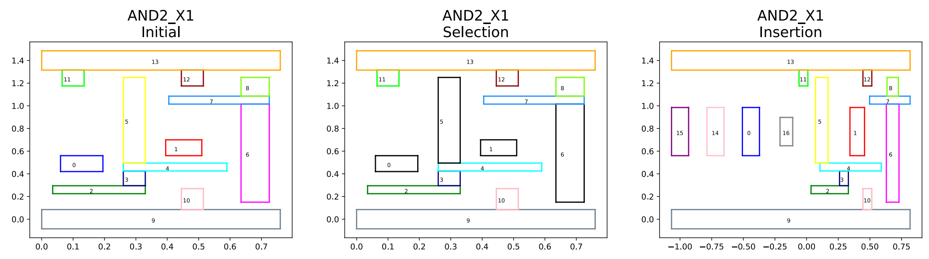
Gate	Gate Sizing	Percentage Increase
AND2	X1	135.86
AND2	X2	88.68
AND2	X4	23.54
AND3	X1	92.37
AND3	X2	57.24
AND3	X4	8.61
AND4	X1	57.24
AND4	X2	42.86
AND4	X4	13.36
AOI21	X1	138.49
AOI21	X2	35.15
AOI21	X4	17.61
AOI211	X1	91.32
AOI211	X2	27.34
AOI211	X4	19.02
AOI22	X1	87.37
AOI22	X2	21.64
AOI22	X4	2.32
AOI221	X1	58.33
AOI221	X2	29.78
AOI221	X4	10.53
AOI222	X1	30.43
AOI222	X2	0
AOI222	X4	0
BUF	X1	214.91
BUF	X2	153.95
BUF	X4	41.35
BUF	X8	0
BUF	X16	0
BUF	X32	35.55
CLKBUF	X1	216.67
CLKBUF	X2	154.61
CLKBUF	X3	88.95
CLKGATE	X8	7.49
CLKGATE	X1	0
CLKGATE	X2	14
CLKGATE	X4	5.34
DFF	X1	6.11
DFF	X2	6.65
DFFR	X1	4.08
DFFR	X2	2.03
DFFRS	X1	5.15
DFFRS	X2	2.28
DFFS	X1	6.51
DFFS	X2	6.7

Table 4.4: Secured Standard Cells in *Nangate FreePDK 45nm* - Part 2 - Percentage increase of the area of each standard cell compared to its non-secure version.

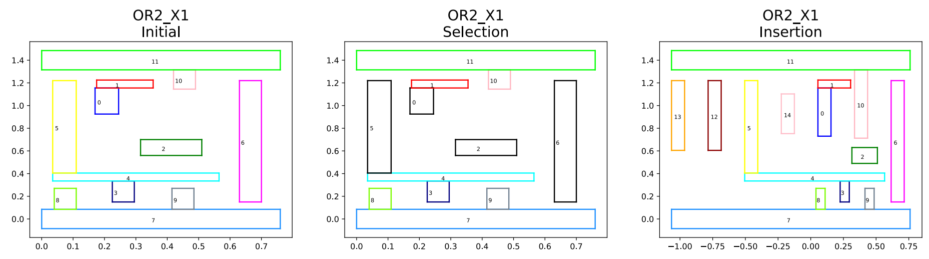
Gate	Gate Sizing	Percentage Increase
DLH	X1	33.68
DLH	X2	25.12
DLL	X1	26.71
DLL	X2	28.83
FA	X1	79.44
HA	X1	33.42
INV	X1	369.08
INV	X2	212.28
INV	X4	87.37
INV	X8	23.39
INV	X16	6.42
INV	X32	0
MUX2	X1	34.77
MUX2	X2	43.71
NAND2	X1	216.67
NAND2	X2	87.37
NAND2	X4	17.84
NAND3	X1	135.86
NAND3	X2	36.47
NAND3	X4	164.98
NAND4	X1	90
NAND4	X2	21.49
NAND4	X4	2.63
NOR2	X1	216.67
NOR2	X2	88.95
NOR2	X4	22.81
NOR3	X1	135.2
NOR3	X2	34.77
NOR3	X4	12.97
NOR4	X1	90
NOR4	X2	14.77
NOR4	X4	12.43
OAI21	X1	137.5
OAI21	X2	40.41
OAI21	X4	24.7
OAI211	X1	87.37
OAI211	X2	19.44
OAI211	X4	0
OAI22	X1	90
OAI22	X2	22.08
OAI22	X4	100.08
OAI221	X1	58.33
OAI221	X2	17.22
OAI221	X4	7.89



(a) INV Gate

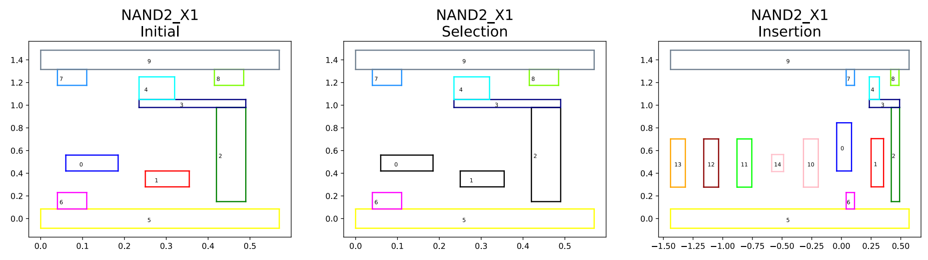


(b) AND Gate

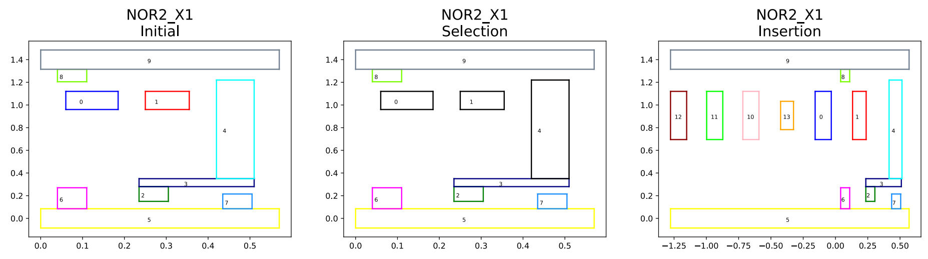


(c) OR Gate

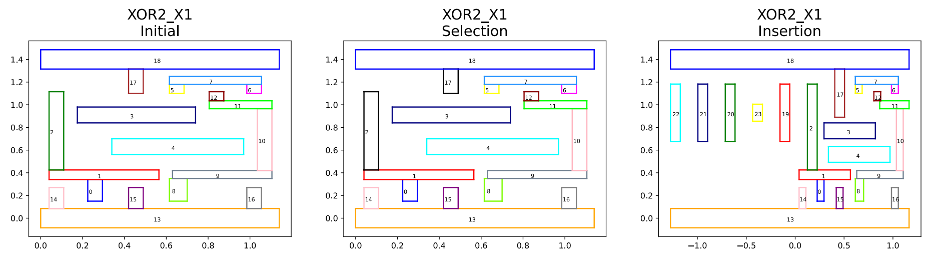
Figure 4·9: End-to-end Insertion Flow - Part 1 - (left) Initial layout, (center) polygons selected for insertion marked in black, (right) after insertion-DRC-fix. We show the results for three of the seven gates: INV_X1, AND2_X1, OR2_X1.



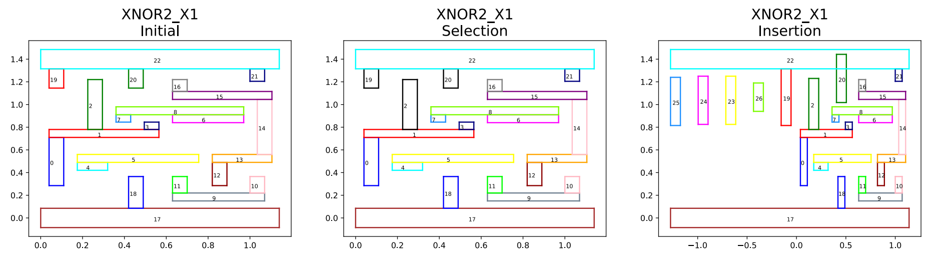
(a) NAND Gate



(b) NOR Gate



(c) XOR Gate



(d) XNOR Gate

Figure 4·10: End-to-end Insertion Flow - Part 2 - (left) Initial layout, (center) polygons selected for insertion marked in black, (right) after insertion-DRC-fix. We show the results for four of the seven gates: NAND2_X1, NOR2_X1, XOR2_X1, and XNOR2_X1.

Table 4.5: Secured Standard Cells in *Nangate FreePDK 45nm* - Part 3 - Percentage increase of the area of each standard cell compared to its non-secure version.

Gate	Gate Sizing	Percentage Increase
OAI222	X1	28.95
OAI222	X2	12.5
OAI222	X4	0
OAI33	X1	48.5
OR2	X1	135.86
OR2	X2	88.68
OR2	X4	14.04
OR3	X1	88.68
OR3	X2	57.24
OR3	X4	12.2
OR4	X1	57.24
OR4	X2	42.86
OR4	X4	14.57
SDFF	X2	6.58
SDFF	X1	1.43
SDFFR	X1	1.26
SDFFRS	X1	5.99
SDFFRS	X2	3.27
SDFFS	X1	10.79
SDFFS	X2	0
TBUF	X1	22.53
TBUF	X2	32.02
TBUF	X4	66.03
TBUF	X8	0.29
TBUF	X16	2.23
TINV	X1	135.86
TLAT	X1	25.91
XNOR2	X1	65.57
XNOR2	X2	126.84
XOR2	X1	76.32
XOR2	X2	30.99

insertion-drc-fix flow to insert the nanoantennas. SecRLCAD was capable of finding insertions both at the edge for some cases, such as XNOR2_X1, and at the center for example in NOR2_X1. This is because, for example, the area around the center is somewhat tight in the XNOR2_X1 therefore SecRLCAD had to shift the horizontal metals to make space around the center. We also see that the spacing constraint and the original connections have also been maintained but have been tightened in order to cause as small of an area increase as possible. We also observe that in many of the layouts there is a close to optimal amount of metal reuse.

Table 4.3, Table 4.4, and Table 4.5 altogether show all the gates in the *Nangate*

FreePDK 45nm and the area comparison of the non-secure and secure standard cells. We observe that INV_X1 gate has a maximum increase of 369%. The average area gain is 56%. The smaller gates will have a larger area increase since these gates inherently have less polygons that can be reused. Therefore INV_X1 having the highest overhead is expected since we have very few metal polygons to use for our structures and have to insert five grating structures. Compared to a potential 100% increase in the dual-gate pairs, this is a major reduction in the overhead. In Table 4.6 we show the area overheads for a variety of hardware blocks for three different standard cell libraries.

Table 4.6: Various IC Chips Using Different Standard Cell Libraries. *Full FreePDK 45nm* is the complete 140 standard cell gates. *Reduced FreePDK 45nm* is the seven small-sized standard cells, INV_X1, AND2_X1, OR2_X1, NAND2_X1, NOR2_X1, XOR2_X1, and XNOR2_X1. We show the normalized area values of each hardware block to the corresponding *Full FreePDK 45nm* version.

Standard Cell Library	Hardware Blocks	Area (μm^2)	Normalized Area Value
Full FreePDK 45nm	Exponential	1208.954	1×
	Max	733.163	1×
	ReLU	196.536	1×
	Add	925.866	1×
	Float Multiplier	6242.691	1×
	Division	18646.663	1×
	Systolic Array 16pe	418279.730	1×
	Sodor 1-Stage	88807.267	1×
	Sodor 3-Stage	99545.198	1×
Reduced FreePDK 45nm	Exponential	1370.39	1.13×
	Max	835.488	1.14×
	ReLU	196.536	1×
	Add	1026.791	1.11×
	Float Multiplier	10257.866	1.64×
	Division	20161.186	1.09×
	Systolic Array 16pe	605135.350	1.44×
	Sodor 1-Stage	114772.579	1.29×
	Sodor 3-Stage	130277.259	1.32×
Nanoantenna-inserted FreePDK 45nm	Exponential	1905.11	1.57×
	Max	1194.00	1.62×
	ReLU	196.53	1×
	Add	1336.78	1.44×
	Float Multiplier	13145.83917	2.07×
	Division	29392.16	1.52×
	Systolic Array 16pe	828091.6585	1.98×
	Sodor 1-Stage	147169.47	1.65×
	Sodor 3-Stage	166124.76	1.66×

Full FreePDK 45nm library has access to all 140 standard cells available in the *Nangate FreePDK 45nm* library. *Reduced FreePDK 45nm* library has access to seven of the standard cells available in the *Nangate FreePDK 45nm* library, INV_X1, AND2_X1, OR2_X1, NAND2_X1, NOR2_X1, XOR2_X1, and XNOR2_X1. *Nanoantenna-inserted FreePDK 45nm* library has access to seven of the standard cells available in the *Nangate FreePDK 45nm* library, INV_X1, AND2_X1, OR2_X1, NAND2_X1, NOR2_X1, XOR2_X1, and XNOR2_X1 but they also have nanoantennas inserted. We observe that the area difference increases for more complicated hardware blocks. The most complicated hardware block we design is the Systolic Array 16pe. For this hardware block, we observe that the area overhead of just reducing the number of gate options is 1.44×. The reason this occurs is because the more complicated hardware designs utilize the more complex standard cells more often compared to simpler designs. Not having access to these standard cells means the synthesis tool has to reconstruct these designs using the simpler standard cells. By inserting nanoantennas into these seven gates, we further increase the area overhead up to 1.98×. The average area overhead of *Reduced FreePDK 45nm* hardware blocks over *Full FreePDK 45nm* is 1.25×, while for *Nanoantenna-inserted FreePDK 45nm* hardware blocks it is 1.6×.

4.4 Summary

This chapter proposed SecRLCAD, an RL-based CAD flow to insert nanoantennas into each individual standard cell in a given standard cell library instead of just non-functional filler cells or dual-gate pairs. We automate the flow end-to-end to take an existing standard cell library as input, insert an optical nanoantenna in each cell of the library, and generate a secure standard cell library as output. Our flow is highly generic and can handle new tech nodes, nanoantenna designs, and standard cell libraries. We automate SecRLCAD to generate all the necessary files for synthesis so that there is minimal overhead and the secured cells are easily used in CAD flows.

Furthermore, we experiment with manually designing seven standard cells for GF 22nm

FD-SOI (GlobalFoundries, 2006). Based on this experience, we can reduce the design time of securing the $\sim$100 standard cell gates from $\sim$28.5 months to only a couple of hours. Utilizing SecRLCAD, we can secure the Nangate FreePDK45nm Library at an average area increase of only 56% compared to the $\sim$ 100% of the prior methods. Finally, it is important to note that given a certain resolution of gates, inserting a nanoantenna could give security at a given fidelity. The resolution depends on how many gates in the IC an attacker would need to create enough space to insert their HT circuit into. Having such a scheme would further reduce the area overhead of the nanoantennas since each functional gate would not need to be a nanoantenna-inserted gate. This is out of the scope of this work but opens interesting possibilities for future work.

Chapter 5

Conclusion and Future Work

This thesis presents and evaluates two thrusts of using ML for hardware design and management In this chapter, we summarize the contributions of this thesis and outline the directions for future work.

5.1 Summary of Major Contributions

On the hardware management front, to improve processor performance, we propose Puppeteer, a low-overhead non-invasive RF-based hardware manager that predicts which prefetchers should be switched ON for a given instruction window. To the best of our knowledge, Puppeteer is the only manager capable of adapting multiple prefetchers at runtime both at the same cache level and across different cache levels. Puppeteer is trained offline and then at runtime utilizes prefetcher invariant events to choose which prefetchers should be switched ON. Puppeteer is trained to maximize processor performance instead of the model accuracy by utilizing a regression scheme instead of classification. Finally, Puppeteer is trained using only a tiny portion ($< 10\%$) of the data and can still achieve high performance. Effectively, it shows good generalization to the underlying distribution with a low amount of data. For the 232 traces that we evaluated, Puppeteer achieves an average performance gain of 46.0% in 1 Core (1C), 25.8% in 4 Core (4C), and 11.9% in 8 Core (8C) processors compared to a system with no prefetching. Furthermore, the main benefit of Puppeteer is its capability to reduce negative outliers. Puppeteer reduces negative outliers by 89%, down to 8 outliers from 53, and a 20% reduction in the IPC loss of the worst-case outlier compared to the state-of-the-art prior prefetcher managers.

On the hardware design front, we propose an RL-based CAD flow, SecRLCAD, which secures standard cell libraries by inserting nanoantennas into the standard cells using a graph-cut algorithm and an RL agent to fix DRC issues. SecRLCAD inserts nanoantennas into individual standard cells and not just into filler cells or into dual-gate pairs. Out methodology allows complete coverage of an IC and decreases the opportunities for a malicious person to insert HTs. Our flow is a novel way to secure standard cells against HT and is faster than the high overhead imaging methods that require IC delayering. SecRLCAD speeds up the design time of secure standard cells by four orders of magnitude as compared to manual design. We build SecRLCAD to be as modular as possible, and it can be readily adapted to support new nanoantenna designs, different technology nodes and different standard cell libraries.

5.2 Future Work

5.2.1 Puppeteer

Puppeteer manages hardware prefetchers across the memory hierarchy. There are multiple immediate ways that Puppeteer can be extended.

Extending Puppeteer to Other Hardware Components

Puppeteer can be extended to manage branch predicting, Dynamic Voltage Frequency Scaling (DVFS) policies, and cache reconfiguration. The idea here would be certain application regions perform better when given a unique combination of certain hardware components, and am application can benefit from different combinations of these components. For example, an application region that has regular branching behavior but has irregular prefetching behavior could benefit from a simple branch predictor such as a one-level branch predictor and an irregular access prefetcher such as a Markov chain prefetcher. Another application region might have regular prefetching behavior but irregular branching behavior, hence, a one-level branch predictor and a Markov chain prefetcher would not give good performance. The optimization space has many local minima and maxima, making it difficult to decide on which components to use in combination together. Sometimes to achieve higher performance, a manager would need to be able to turn OFF certain components that would for the average case make sense to leave ON. At the simplest case, consider the case of a single branch predictor (BP) and a single prefetcher (PF). There would be four unique management configurations where (i) BP = ON, PF = ON, (ii) BP = ON, PF = OFF, (iii) BP = OFF, PF = ON, and (iv) BP = OFF, PF = OFF.

The complex interactions in a state-of-the-art out-of-order (OoO) core make it difficult to say that all application regions would benefit from either/or both the BP and the PF being left ON indefinitely. Therefore, having an ML-based manager for multiple components could benefit performance. To the best of our knowledge, there are no works that have cross-component managers.

Extending Puppeteer to Multi-Threaded Processors

Puppeteer was developed for single-core and many-core processors. The current implementation does not take multi-threading into account. Multi-threading brings other new interesting challenges to Puppeteer. The resource sharing that exists in multi-core processors is still somewhat abstracted from the short-term behavior recorded during the scale of an instruction window. This is because resource sharing events at high-level memory such as L3 and LLC occurs less frequently during 100k instructions. Therefore, in the current Puppeteer we can use a single core model to achieve reasonably good performance in the multi-core experiments. But with multi-threading, this assumption no longer holds. From the perspective of each thread, there are far more non-deterministic events, such as L1 data traffic, occurring on each core because all the hardware is now shared with at least one other thread. This would mean that resource sharing would need to be accounted for in a better manner in Puppeteer. The ML model could require a "hardware-resource-invariant", i.e., application-related class of features, e.g., the number of unique addresses accessed, to guide decision-making.

Extending Puppeteer to Other Hardware Optimization Goals

We design Puppeteer to achieve the highest processor performance possible. The corollary would be to look into secondary goals such as power or bandwidth utilization. Prefetchers spend around 5-10% of the total power of a computing system (Kamruzzaman et al., 2011; Jiménez et al., 2014; Kalani and Panda, 2021). Logically, there are application regions that do not have much performance to gain from certain prefetchers being turned ON, but they would still needlessly spend power. By turning these prefetchers OFF, we could utilize the now free power in other hardware components or even to scale frequency. Additional hardware optimization goals, such as reducing power and memory utilization, could be coupled with using different hardware components, such as branch predicting and DVFS, and utilized with multi-threading to accomplish complex balancing among these components to achieve even higher performance and lower power. Effectively the goal here would be to allocate the power to the hardware component that would utilize it the best for a given application region.

5.2.2 SecRLCAD

SecRLCAD has been developed to take a known standard cell layout and embed the nanoantenna design into it. Two improvements could be made upon this methodology.

Extending SecRLCAD to Design Standard Cells from Scratch

Instead of taking a known standard cell design and embedding additional metals corresponding to the optical nanoantenna structure, we can develop a methodology to design the standard cell design around the nanoantenna structure. Ren et al. (Ren and Fojtik, 2021) suggest using genetic algorithms to build standard cells and apply an RL agent to clean the DRC issues on the metal layer. Our approach would be similar in flow but would have the additional constraint of sustaining the optical nanoantenna structure inside the layout built by the algorithmic flow. The immediate benefit of such a flow compared to the current SecRLCAD approach is this approach would be able to reuse more of the optical nanoantenna structures for cell functionality. This would result in lower standard area compared to the area of the current SecRLCAD approach. We can also include power and performance as optimization sub-goals into the reward function which would improve performance and lower power.

Extending SecRLCAD to Design Standard Cells and Nanoantennas from Scratch

The natural next step would be to eliminate the known nanoantenna design and start from no known design. We can envision a flow where an algorithm, while building the standard cells, checks for optical properties and builds the standard cell to generate the required optical signature. This type of approach would lead to non-intuitive nanoantenna designs. The area overhead would be further reduced compared to the prior approaches. Power and performance numbers would also be better since the design would be built without restrictive structures. The main challenge with this approach stems from achieving high accuracy in optical simulations in place of FDTD simulations using a NN-based approach.

One would need to check the optical properties of the layouts while designing the standard cells. Unfortunately, FDTD simulations take an extremely long time ($\sim$ 24 hours). The length of time required for a single FDTD simulation would prevent an algorithm from checking for the signal in an iterative manner. Therefore, to accomplish this future direction, there needs to be another algorithm that quickly converges to the expected optical response for a given structure. Potentially, this could be done using a Generative NN approach.

5.3 Final Remarks

In summary, we propose, implement, and evaluate two ML-based approaches for hardware optimization and management. Our first approach, Puppeteer, manages and improves the performance of hardware prefetchers at run time using an RF algorithm. Our second approach, SecRLCAD, utilizes RL and a graph-cut algorithm to insert optical nanoantennas into standard cells to secure them against HT. We strongly believe that ML will aid hardware designers in coming up with better designs, which will have better performance, lower power, lower area and/or costs.

References

Aamodt, T. M., Fung, W. W. L., and Rogers, T. G. (2018). General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140.

Adato, A. R., Uyar, A., Zangeneh, M., Zhou, B., Joshi, A., Goldberg, B., and Unlu, M. S. (2016). Rapid mapping of digital integrated circuit logic gates via multi-spectral backside imaging. *arXiv preprint arXiv:1605.09306*.

Adato, F. . R., Uyar, A., Zangeneh, M., Zhou, B., Joshi, A., Goldberg, B., and Unlu, S. (2015). Integrated nanoantenna labels for rapid security testing of semiconductor circuits. In *Frontiers in Optics*, pages FTh1B–2. Optical Society of America.

Al-Zoubi, H., Milenkovic, A., and Milenkovic, M. (2004). Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272.

Alpha, S. (2021). Deep dive into semiconductor industry. https://seekingalpha.com/article/4473172-deep-dive-semiconductor-industry.

AMD (2021a). Amd to acquire xilinx. https://www.amd.com/en/corporate/xilinx-acquisition.

AMD (2021b). Nvidia acquires arm. https://arm.nvidia.com/.

(AMD), A. M. D. (2017). AMD Ryzen Processor. https://www.amd.com/en/ryzen.

AnalyticsVidhya (2021). How machine learning hardware and algorithms power apples latest watch and iphones. https://www.analyticsvidhya.com/blog/2018/09/how-machine-learning-hardware-and-algorithms-power-apples-latest-watch-and-iphones/.

Arel, I., Liu, C., Urbanik, T., and Kohls, A. G. (2010). Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135.

Asadizanjani, N., Tehranipoor, M., and Forte, D. (2017). Counterfeit electronics detection using image processing and machine learning. In *Journal of physics: conference series*, volume 787, page 012023. IOP Publishing.

Bakhshalipour, M., Shakerinava, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. (2019). Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411. IEEE.

Becker, G. T., Regazzoni, F., Paar, C., and Burleson, W. P. (2013). Stealthy dopant-level hardware trojans. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 197–214. Springer.

Bera, R., Kanellopoulos, K., Nori, A. V., Shahroodi, T., Subramoney, S., and Mutlu, O. (2021). Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*.

Bhatia, E., Chacon, G., Pugsley, S., Teran, E., Gratz, P. V., and Jiménez, D. A. (2019). Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE.

Biau, G. and Scornet, E. (2016). A random forest guided tour. *TEST*, 25(2):197–227. https://doi.org/10.1007/s11749-016-0481-7.

Bios., A. M. D. (2010). kernel developer guide (bkdg) for AMD family 10h models 00h-0fh processors. https://www.amd.com/system/files/TechDocs/31116.pdf.

Black, B., Annavaram, M., Brekelbaum, N., DeVale, J., Jiang, L., Loh, G. H., McCaule, D., Morrow, P., Nelson, D. W., Pantuso, D., et al. (2006). Die stacking (3d) microarchitecture. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 469–479. IEEE.

Black, D. L., Rashid, R. F., Golub, D. B., and Hill, C. R. (1989). Translation lookaside buffer consistency: A software approach. *ACM SIGARCH Computer Architecture News*, 17(2):113–122.

Bloomberg (2020). Car chip fabrication cost. https://tinyurl.com/2kazk3c7.

Botero, U. J., Wilson, R., Lu, H., Rahman, M. T., Mallaiyan, M. A., Ganji, F., Asadizanjani, N., Tehranipoor, M. M., Woodard, D. L., and Forte, D. (2021). Hardware trust and assurance through reverse engineering: A tutorial and outlook from image analysis and machine learning perspectives. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(4):1–53.

Braun, P. and Litz, H. (2019). Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Bu, X., Rao, J., and Xu, C.-Z. (2009). A reinforcement learning approach to online web systems auto-configuration. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 2–11. IEEE.

Cadence (2022a). Cadence liberate. `https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/library-characterization/liberate-characterization.html`.

Cadence (2022b). Virtuoso abstract generator. `https://www.cadence.com/en_US/home/training/all-courses/86197.html`.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1997). Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222.

Cantin, J. F., Lipasti, M. H., and Smith, J. E. (2006). Stealth prefetching. *ACM Sigplan Notices*, 41(11):274–282.

Carter, R., Mazurier, J., Pirro, L., Sachse, J., Baars, P., Faul, J., Grass, C., Grasshoff, G., Javorka, P., Kammler, T., et al. (2016). 22nm fdsoi technology for emerging mobile, internet-of-things, and rf applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2–2. IEEE.

Cheng, R. and Yan, J. (2021). On joint learning for solving placement and routing in chip design. *Advances in Neural Information Processing Systems*, 34.

CNBC (2021a). Security concerns on global semiconductor shortage. `https://www.cnbc.com/2021/08/04/moodys-analytics-on-global-semiconductor-shortage-and-governments.html`.

CNBC (2021b). Tsmc arizona plant. [online] `https://www.cnbc.com/2021/10/16/tsmc-taiwanese-chipmaker-ramping-production-to-end-chip-shortage.html#:~:text=TSMC's%20%2412%20billion%205%2Dnanometer,%2C%20on%20September%2028%2C%202021.&text=TSMC%20alone%20was%20responsible%20for,2019%2C%20according%20to%20the%20company.` (Accessed on 02/09/2021).

Cochran, R. et al. (2011). Pack & cap: adaptive dvfs and thread packing under power caps. In *Proc. MICRO*, pages 175–185.

Corporation, S. P. E. (2006). SPEC CPU® 2006. `https://www.spec.org/cpu2006/`.

Corporation, S. P. E. (2017). SPEC CPU® 2017. `https://www.spec.org/cpu2017/`.

Coskun, A., Eris, F., Joshi, A., Kahng, A. B., Ma, Y., Narayan, A., and Srinivas, V. (2020). Cross-layer co-optimization of network design and chiplet placement in 2.5-d systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):5183–5196.

Coskun, A., Eris, F., Joshi, A., Kahng, A. B., Ma, Y., and Srinivas, V. (2018). A cross-layer methodology for design and optimization of networks in 2.5 d systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99. IEEE.

Dai, Z., Liu, H., Le, Q., and Tan, M. (2021). Coatnet: Marrying convolution and attention for all data sizes. *Advances in Neural Information Processing Systems*, 34.

Dally, W. J., Turakhia, Y., and Han, S. (2020). Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57.

Dean, J. (2020). 1.1 the deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 8–14. IEEE.

Dean, J., Patterson, D., and Young, C. (2018). A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29.

Demirkiran, C., Eris, F., Wang, G., Elmhurst, J., Moore, N., Harris, N. C., Basumallik, A., Janapa Reddi, V., Joshi, A., and Bunandar, D. (2021). An electro-photonic system for accelerating deep neural networks. *arXiv e-prints*, pages arXiv–2109.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570.

Dong, C., Xu, Y., Liu, X., Zhang, F., He, G., and Chen, Y. (2020). Hardware trojans in chips: a survey for detection and prevention. *Sensors*, 20(18):5165.

Dong, M. J., Yung, K. G., and Kaiser, W. J. (1997). Low power signal processing architectures for network microsensors. In *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pages 173–177. IEEE.

Drive Tesla Canada (2021). Samsung fsd 4.0. [online] `https://driveteslacanada.ca/news/samsung-to-make-teslas-hw-4-0-full-self-driving-fsd-chip-report/`. (Accessed on 12/27/2021).

Du, S., Huang, T., Hou, J., Song, S., and Song, Y. (2019). Fpga based acceleration of game theory algorithm in edge computing for autonomous driving. *Journal of Systems Architecture*, 93:33–39.

Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. (2010). Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346.

Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. (2011). Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 39(3):141–152.

Ebrahimi, E., Mutlu, O., Lee, C. J., and Patt, Y. N. (2009a). Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326. IEEE/ACM.

Ebrahimi, E., Mutlu, O., and Patt, Y. N. (2009b). Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 7–17. IEEE.

Eris, F., Joshi, A., Kahng, A. B., Ma, Y., Mojumder, S., and Zhang, T. (2018). Leveraging thermally-aware chiplet organization in 2.5 d systems to reclaim dark silicon. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1441–1446. IEEE.

Falsafi, B. and Wenisch, T. F. (2014). A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67.

Fedchenko, V., Neglia, G., and Ribeiro, B. (2019). Feedforward neural networks for caching: N enough or too much? *ACM SIGMETRICS Performance Evaluation Review*, 46(3):139–142.

Forbes (2021). 2022 computing trends. [online] `https://www.forbes.com/sites/bernardmarr/2021/10/25/the-5-biggest-cloud-computing-trends-in-2022/?sh=5e2fc0e32267/`. (Accessed on 12/27/2021).

Fortune (2021). Bitcoin energy consumption. [online] `https://fortune.com/2021/10/26/bitcoin-electricity-consumption-carbon-footprin/`. (Accessed on 12/27/2021).

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., and Pineau, J. (2018). An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*.

Ganusov, I. and Burtscher, M. (2005). Future execution: A hardware prefetching technique for chip multiprocessors. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 350–360. IEEE.

Gdspy (2022). Gdspy documentation. https://gdspy.readthedocs.io/en/stable/.

GF (2021). Gf announcement. https://www.cnbc.com/2021/10/30/globalfoundries-ceo-were-sold-out-of-semiconductor-chip-capacity-through-2023.html.

GlobalFoundries (2006). Gf 22nm fd-soi. [online] https://www.globalfoundries.com/sites/default/files/22fdx-product-brief.pdf. (Accessed on 02/09/2022).

Gomez, F. J., Burger, D., and Miikkulainen, R. (2001). A neuro-evolution method for dynamic resource allocation on a chip multiprocessor. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 4, pages 2355–2360. IEEE.

Guo, P., Hou, W., Guo, L., Sun, W., Liu, C., Bao, H., Duong, L. H., and Liu, W. (2019). Fault-tolerant routing mechanism in 3d optical network-on-chip based on node reuse. *IEEE Transactions on Parallel and Distributed Systems*, 31(3):547–564.

Hasegawa, K., Yanagisawa, M., and Togawa, N. (2017). Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.

Hashemi, M. et al. (2018). Learning memory access patterns. *arXiv:1803.02329*.

HBR (2021). How much energy does bitcoin actually consume. [online] https://hbr.org/2021/05/how-much-energy-does-bitcoin-actually-consume/. (Accessed on 12/27/2021).

Heirman, W., Bois, K. D., Vandriessche, Y., Eyerman, S., and Hur, I. (2018). Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–11.

Hempstead, M., Tripathi, N., Mauro, P., Wei, G.-Y., and Brooks, D. (2005). An ultra low power system architecture for sensor network applications. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 208–219. IEEE.

Hiebel, J., Brown, L. E., and Wang, Z. (2019). Machine learning for fine-grained hardware prefetcher control. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–9.

Hospodar, G., Gierlichs, B., De Mulder, E., Verbauwhede, I., and Vandewalle, J. (2011). Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293–302.

HPE, Cray, AMD (2021). Exascale computer. [online] https://www.hpe.com/us/en/compute/hpc/supercomputing/cray-exascale-supercomputer.html?jumpid=ps_4ha3v2drf6_aid-520061464&ef_id=EAIaIQobChMI6qOehbWD9QIVYsmUCR1hwgHUEAAYASAAEgJZdPD_BwE:G:s&s_kwcid=AL!13472!3!523569994773!e!!g!!exascale%20computing!13236197030!123093432656&. (Accessed on 12/27/2021).

Hu, W., Mao, B., Oberg, J., and Kastner, R. (2016). Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):44–52.

Huang, Z., Wang, Q., Chen, Y., and Jiang, X. (2020). A survey on machine learning against hardware trojan attacks: Recent advances and challenges. *IEEE Access*, 8:10796–10826.

Hur, I. and Lin, C. (2006). Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408. IEEE.

Inquirer (2017). 3nm fabrication cost. https://tinyurl.com/6m8xncww.

Intel (2011). Intel® 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2. https://docplayer.net/7459362-Intel-64-and-ia-32-architectures-software-developer-s-manual.html.

Intel (2017). Intel i9. https://www.intel.com/content/www/us/en/products/details/processors/core/i9.html.

Ipek, E., Mutlu, O., Martínez, J. F., and Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. *ACM SIGARCH Computer Architecture News*, 36(3):39–50.

Ippolito, S., Thorne, S., Eraslan, M., Goldberg, B., Ünlü, M., and Leblebici, Y. (2004). High spatial resolution subsurface thermal emission microscopy. *Applied Physics Letters*, 84(22):4529–4531.

Jacob, N., Merli, D., Heyszl, J., and Sigl, G. (2014). Hardware trojans: current challenges and approaches. *IET Computers & Digital Techniques*, 8(6):264–273.

Jain, R., Panda, P. R., and Subramoney, S. (2016). Machine learned machines: Adaptive co-optimization of caches, cores, and on-chip network. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 253–256. IEEE.

Jiang, Z., Songhori, E., Wang, S., Goldie, A., Mirhoseini, A., Jiang, J., Lee, Y.-J., and Pan, D. Z. (2021). Delving into macro placement with reinforcement learning. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–3. IEEE.

Jiménez, D. A. and Teran, E. (2017). Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448. IEEE.

Jiménez, V., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., and Mealey, B. G. (2014). Adaptive prefetching on power7: improving performance and power consumption. *ACM Transactions on Parallel Computing (TOPC)*, 1(1):1–25.

Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and O'Connell, F. P. (2012). Making data prefetch smarter: Adaptive prefetching on power7. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 137–146. IEEE.

Jin, Y., Maliuk, D., and Makris, Y. (2012). Post-deployment trust evaluation in wireless cryptographic ics. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 965–970. IEEE.

Joshi, A., Batten, C., Kwon, Y.-J., Beamer, S., Shamim, I., Asanovic, K., and Stojanovic, V. (2009). Silicon-photonic clos networks for global on-chip communication. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 124–133. IEEE.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

Jurcevic, P., Javadi-Abhari, A., Bishop, L. S., Lauer, I., Bogorin, D. F., Brink, M., Capelluto, L., Günlük, O., Itoko, T., Kanazawa, N., et al. (2021). Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology*, 6(2):025020.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

Kagi, A., Goodman, J. R., and Burger, D. (1996). Memory bandwidth limitations of future microprocessors. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 78–78. IEEE.

Kalani, N. S. and Panda, B. (2021). Instruction criticality based energy-efficient hardware data prefetching. *IEEE Computer Architecture Letters*, 20(2):146–149.

Kamruzzaman, M., Swanson, S., and Tullsen, D. M. (2011). Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 393–404.

Kang, H. and Wong, J. L. (2013). To hardware prefetch or not to prefetch?: a virtualized environment study and core binding approach. In *ACM SIGPLAN Notices*, volume 48:4, pages 357–368.

Karri, R., Rajendran, J., Rosenfeld, K., and Tehranipoor, M. (2010). Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46.

Kim, J., Pugsley, S. H., Gratz, P. V., Reddy, A. N., Wilkerson, C., and Chishti, Z. (2016). Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE.

Kim, J., Teran, E., Gratz, P. V., Jiménez, D. A., Pugsley, S. H., and Wilkerson, C. (2017). Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices*, 52(4):737–749.

Klayout (2022). Klayout documentation. `https://www.klayout.de/`.

Knudsen, J. (2008). Nangate 45nm open cell library. *CDNLive, EMEA*.

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*, pages 1–19. IEEE.

Köklü, F. H. and Ünlü, M. S. (2010). Subsurface microscopy of interconnect layers of an integrated circuit. *Optics letters*, 35(2):184–186.

Kondguli, S. and Huang, M. (2018). Division of labor: A more effective approach to prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–95. IEEE.

Kulkarni, V. Y. and Sinha, P. K. (2012). Pruning of random forest classifiers: A survey and future directions. In *2012 International Conference on Data Science & Engineering (ICDSE)*, pages 64–68. IEEE.

Kursa, M. B. (2014). Robustness of random forest-based gene selection methods. *BMC bioinformatics*, 15(1):1–8.

Lee, J., Kim, H., and Vuduc, R. (2012). When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):1–29.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

Li, J., Ni, L., Chen, J., and Zhou, E. (2016a). A novel hardware trojan detection based on bp neural network. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2790–2794. IEEE.

Li, S., Xu, C., Zou, Q., Zhao, J., Lu, Y., and Xie, Y. (2016b). Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6.

Liao, S.-w., Hung, T.-H., Nguyen, D., Chou, C., Tu, C., and Zhou, H. (2009). Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10.

Lin, L., Kasper, M., Güneysu, T., Paar, C., and Burleson, W. (2009). Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 382–395. Springer.

Lin, Y., Jiang, Z., Gu, J., Li, W., Dhar, S., Ren, H., Khailany, B., and Pan, D. Z. (2020). Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(4):748–761.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: reading kernel memory from user space. In *Proceedings of the USENIX Security Symposium (Security'18)*, pages 973–990. USENIX Association.

Liu, P., Yu, J., and Huang, M. C. (2016). Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–25.

Ma, Y., Delshadtehrani, L., Demirkiran, C., Abellán, J. L., and Joshi, A. (2021). Tap-2.5 d: A thermally-aware chiplet placement methodology for 2.5 d systems. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1246–1251. IEEE.

Maas, M. (2020). A taxonomy of ml for systems problems. *IEEE Micro*, 40(05):8–16.

Maldikar, P. (2014). *Adaptive cache prefetching using machine learning and monitoring hardware performance counters*. PhD thesis, University of Minnesota.

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., et al. (2021). A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Moore (2019). Moore's law is dead. [online] https://towardsdatascience.com/moores-law-is-dead-678119754571#:~:text=It%20was%20calculated%20that%20solely,the%20last%20process%20technology%20node. (Accessed on 02/09/2021).

Moreira, F. B., Diener, M., Navaux, P. O., and Koren, I. (2017). Data mining the memory access stream to detect anomalous application behavior. In *Proceedings of the Computing Frontiers Conference*, pages 45–52.

Mukundan, J. and Martinez, J. F. (2012). Morse: Multi-objective reconfigurable self-optimizing memory scheduler. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE.

Naffziger, S., Beck, N., Burd, T., Lepak, K., Loh, G. H., Subramony, M., and White, S. (2021). Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–70. IEEE.

Nakamura, T., Koizumi, T., Degawa, Y., Irie, H., Sakai, S., and Shioya, R. (2020). D-jolt: Distant jolt prefetcher. *The 1st Instruction Prefetching Championship (IPC1)*.

Narayan, A., Joshi, A., and Coskun, A. K. (2021). System-level management of silicon-photonic networks in 2.5 d systems. In *Silicon Photonics for High-Performance Computing and Beyond*, pages 71–88. CRC Press.

Narayanan, A., Verma, S., Ramadan, E., Babaie, P., and Zhang, Z.-L. (2018). Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53.

Nowroz, A. N., Hu, K., Koushanfar, F., and Reda, S. (2014). Novel techniques for high-sensitivity hardware trojan detection using thermal and power maps. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(12):1792–1805.

Oshiro, T. M., Perez, P. S., and Baranauskas, J. A. (2012). How many trees in a random forest? In *International workshop on machine learning and data mining in pattern recognition*, pages 154–168. Springer.

Pakalapati, S. and Panda, B. (2020). Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–131. IEEE.

Pan, D. (2021). Closing the virtuous cycle of ai for ic and ic for ai. https://ieee-ceda.org/virtual-dlp.

Peled, L., Mannor, S., Weiser, U., and Etsion, Y. (2015). Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 285–297. IEEE.

Peled, L., Weiser, U., and Etsion, Y. (2019). A neural network prefetcher for arbitrary memory access patterns. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–27.

Perelman, E., Hamerly, G., and Calder, B. (2003). Picking statistically valid and early simulation points. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255. IEEE.

Post, F. (2018). Cyber attack on us institutions. firstpost.com/tech/news-analysis/cyber-attack-using-microchips-is-the-latest-tactic-for-state-sponsored-chinese-hackers-5323751.html.

Pugsley, S. et al. (2019). DPC3. https://dpc3.compas.cs.stonybrook.edu/.

Pugsley, S. et al. (2020a). Apple M1 Pro and M1 Max. https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/#:~:text=M1%20Pro%20features%2033.7%20billion,64GB%20of%20fast%20unified%20memory.

Pugsley, S. et al. (2020b). ChampSim. https://github.com/ChampSim/ChampSim.

Pugsley, S. et al. (2020c). IPC1. https://research.ece.ncsu.edu/ipc/.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research.*

Rahman, S., Burtscher, M., Zong, Z., and Qasem, A. (2015). Maximizing hardware prefetch effectiveness with machine learning. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 383–389. IEEE.

Ravi, G. S. and Lipasti, M. H. (2017). Charstar: Clock hierarchy aware resource scaling in tiled architectures. *ACM SIGARCH Computer Architecture News*, 45(2):147–160.

Ren, H. and Fojtik, M. (2021). Nvcell: Standard cell layout in advanced technology nodes with reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1291–1294. IEEE.

Ren, H., Godil, S., Khailany, B., Kirby, R., Liao, H., Nath, S., Raiman, J., and Roy, R. (2021). Optimizing vlsi implementation with reinforcement learning-iccad special session paper. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–6. IEEE.

Research Markets (2022). End user computing market report. [online] https://www.researchandmarkets.com/reports/4621525/end-user-computing-market-by-solution-service#:~:text=The%20global%20End%20User%20Computing,12.8%25%20during%20the%20forecast%20period. (Accessed on 02/09/2022).

Reuters (2022). Intel's $20 bln ohio factory could become world's largest chip plant. [online] https://www.reuters.com/technology/intel-plans-new-chip-manufacturing-site-ohio-report-2022-01-21/. (Accessed on 02/05/2022).

Reza, M. F., Le, T. T., De, B., Bayoumi, M., and Zhao, D. (2018). Neuro-noc: Energy optimization in heterogeneous many-core noc using neural networks in dark silicon era. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Rieck, K., Trinius, P., Willems, C., and Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668.

Riquelme, C., Puigcerver, J., Mustafa, B., Neumann, M., Jenatton, R., Susano Pinto, A., Keysers, D., and Houlsby, N. (2021). Scaling vision with sparse mixture of experts. *Advances in Neural Information Processing Systems*, 34.

Rogers, J. (2019). Effects of an lstm composite prefetcher, M.S. thesis, Uppsala Universitet. http://uu.diva-portal.org/smash/get/diva2:1369282/FULLTEXT01.pdf.

Ros, A. and Jimborean, A. (2020). The entangling instruction prefetcher. *IEEE Computer Architecture Letters*, 19(2):84–87.

Rostami, M., Koushanfar, F., and Karri, R. (2014). A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295.

Seznec, A. (2020). The FNL+ MMA Instruction Cache Prefetcher, IPC-1-First Instruction Prefetching Championship. https://hal.inria.fr/hal-02884880.

Shakerinava, M., Bakhshalipour, M., Lotfi-Kamran, P., and Sarbazi-Azad, H. (2019). Multi-lookahead offset prefetching, DPC-3 The Third Data Prefetching Championship. https://mshakerinava.github.io/papers/mlop-dpc3.pdf.

Shevgoor, M., Koladiya, S., Balasubramonian, R., Wilkerson, C., Pugsley, S. H., and Chishti, Z. (2015). Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 141–152. IEEE.

Shi, W., Wang, H., Gu, J., Liu, M., Pan, D. Z., Sun, N., et al. (2021a). RoDesigner: Variation-Aware Optimization for Robust Analog Design with Multi-Task RL. https://openreview.net/pdf?id=8dF_13D2SmD.

Shi, Z. et al. (2019). Applying deep learning to the cache replacement problem. In *Proc. MICRO*, pages 413–425.

Shi, Z., Jain, A., Swersky, K., Hashemi, M., Ranganathan, P., and Lin, C. (2021b). A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 861–873.

Siemens (2022a). Calibre documentation. https://eda.sw.siemens.com/en-US/ic/calibre-design/physical-verification/.

Siemens (2022b). Fill cell leakage. https://blogs.sw.siemens.com/calibre/2016/04/05/reducing-post-placement-leakage-with-stress-enhanced-fill-cells/.

Siemens (2022c). Pex software. https://eda.sw.siemens.com/en-US/ic/calibre-design/circuit-verification/xrc/.

Silva, B. d., Segers, L., Braeken, A., Steenhaut, K., and Touhafi, A. (2018). A low-power fpga-based architecture for microphone arrays in wireless sensor networks. In *International Symposium on Applied Reconfigurable Computing*, pages 281–293. Springer.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676):354–359.

Smith, J. E. (1998). A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215.

Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A. (2006). Spatial memory streaming. *ACM SIGARCH Computer Architecture News*, 34(2):252–263.

Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N. (2007). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74. IEEE.

Srivastava, A., Lazaris, A., Brooks, B., Kannan, R., and Prasanna, V. K. (2019). Predicting memory accesses: the road to compact ml-driven prefetcher. In *Proceedings of the International Symposium on Memory Systems*, pages 461–470.

Steinberg, D. (2009). Cart: classification and regression trees. In *The top ten algorithms in data mining*, pages 193–216. Chapman and Hall/CRC.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Talpes, E., Sarma, D. D., Venkataramanan, G., Bannon, P., McGee, B., Floering, B., Jalote, A., Hsiong, C., Arora, S., Gorti, A., et al. (2020). Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35.

Tarsa, S. J., Lin, C.-K., Keskin, G., Chinya, G., and Wang, H. (2019). Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*.

Tech, E. (2019). As chip design costs skyrocket, 3nm process node is in jeopardy - extremetech. https://bit.ly/2IrGYAC. (Accessed on 02/06/2019).

TechHQ (2021). Where are we with the semiconductor supply chain crises? https://techhq.com/2021/12/where-are-we-with-the-semiconductor-supply-chain-crisis/.

Tehranipoor, M. and Koushanfar, F. (2010). A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25.

Tekeste, T., Saleh, H., Mohammad, B., and Ismail, M. (2018). Ultra-low power qrs detection and ecg compression architecture for iot healthcare devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(2):669–679.

Tesauro, G. (2007). Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30.

Tran, H.-N. and Cambria, E. (2018). A survey of graph processing on graphics processing units. *The Journal of Supercomputing*, 74(5):2086–2115.

Ustun, E., Deng, C., Pal, D., Li, Z., and Zhang, Z. (2020). Accurate operation delay prediction for fpga hls using graph neural networks. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9.

Vanderwiel, S. P. and Lilja, D. J. (2000). Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199. https://doi.org/10.1145/358923.358939.

Vengerov, D. (2009). A reinforcement learning framework for utility-based scheduling in resource-constrained systems. *Future Generation Computer Systems*, 25(7):728–736.

Wang, T., Payberah, A. H., Hagos, D. H., and Vlassov, V. (2022). Accelerate model parallel training by using efficient graph traversal order in device placement. *arXiv preprint arXiv:2201.09676*.

Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. (2003). Guided region prefetching: A cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 388–398. IEEE.

Wei, S., Li, K., Koushanfar, F., and Potkonjak, M. (2012). Hardware trojan horse benchmark via optimal creation and placement of malicious circuitry. In *Proceedings of the Design Automation Conference (DAC)*, pages 90–95. ACM.

Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A. (2009). Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 79–90. IEEE.

Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely Jr, S. C., and Emer, J. (2011). Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441.

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

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24. https://doi.org/10.1145/216585.216588.

Xiao, K., Forte, D., Jin, Y., Karri, R., Bhunia, S., and Tehranipoor, M. (2016). Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–23.

Xu, Q., Geng, H., Chen, S., Yuan, B., Zhuo, C., Kang, Y., and Wen, X. (2021a). Goodfloorplan: Graph convolutional network and reinforcement learning based floorplanning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Xu, W., Song, H., Hou, L., Zheng, H., Zhang, X., Zhang, C., Hu, W., Wang, Y., and Liu, B. (2021b). Soda: Similar 3d object detection accelerator at network edge for autonomous driving. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE.

Xue, M., Gu, C., Liu, W., Yu, S., and O'Neill, M. (2020). Ten years of hardware trojans: a survey from the attacker's perspective. *IET Computers & Digital Techniques*, 14(6):231–246.

Yang, K., Hicks, M., Dong, Q., Austin, T., and Sylvester, D. (2016). A2: Analog malicious hardware. In *2016 IEEE symposium on security and privacy (SP)*, pages 18–37. IEEE.

Yigitbasi, N., Willke, T. L., Liao, G., and Epema, D. (2013). Towards machine learning-based auto-tuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 11–20. IEEE.

Yin, J., Sethumurugan, S., Eckert, Y., Patel, C., Smith, A., Morton, E., Oskin, M., Jerger, N. E., and Loh, G. H. (2020). Experiences with ml-driven design: A noc case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648. IEEE.

Yu, C., Xiao, H., and De Micheli, G. (2018). Developing synthesis flows without human knowledge. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6.

Yu, C. and Zhou, W. (2020). Decision making in synthesis cross technologies using lstms and transfer learning. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 55–60.

Zaraee, N., Zhou, B., Vigil, K., Shahjamali, M. M., Joshi, A., and Ünlü, M. S. (2020). Gate-level validation of integrated circuits with structured-illumination read-out of embedded optical signatures. *IEEE Access*, 8:70900–70912.

Zeng, Y. and Guo, X. (2017). Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 305–311.

Zhang, N., Zheng, K., and Tao, M. (2018). Using grouped linear prediction and accelerated reinforcement learning for online content caching. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE.

Zhang, P., Srivastava, A., Brooks, B., Kannan, R., and Prasanna, V. K. (2020a). Raop: Recurrent neural network augmented offset prefetcher. In *The International Symposium on Memory Systems*, pages 352–362.

Zhang, Y., Ren, H., and Khailany, B. (2020b). Grannite: Graph neural network inference for transferable power estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Zhou, B., Aksoylar, A., Vigil, K., Adato, R., Tan, J., Goldberg, B., Ünlü, M. S., and Joshi, A. (2020). Hardware trojan detection using backside optical imaging. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(1):24–37.

Zhou, B., Gupta, A., Jahanshahi, R., Egele, M., and Joshi, A. (2018). Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 457–468.

Zhou, D. . B., Adato, R., Zangeneh, M., Yang, T., Uyar, A., Goldberg, B., Unlu, S., and Joshi, A. (2015). Detecting hardware trojans using backside optical imaging of embedded watermarks. In *Proceedings of the 52nd Design Automation Conference*, page 111. ACM.

Zhou, Z., Li, X., and Zare, R. N. (2017). Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, 3(12):1337–1344.

CURRICULUM VITAE

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]