APPROXIMATE COMPUTING: FROM CIRCUITS TO SOFTWARE

by

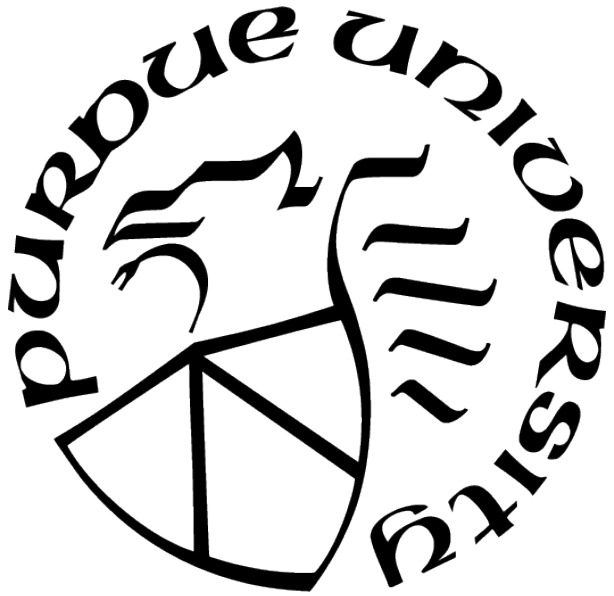
Younghoon Kim

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Anand Raghunathan, Chair

School of Electrical and Computer Engineering

Dr. Cheng-Kok Koh

School of Electrical and Computer Engineering

Dr. Kaushik Roy

School of Electrical and Computer Engineering

Dr. Vijay Raghunathan

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitri Peroulis

ACKNOWLEDGMENTS

My long journey for a Ph.D. degree would have not been possible without the help of great people around me. Firstly, I'd like to express my sincerest gratitude to my advisor, Prof. Anand Raghunathan, for all the support and guidance on my research. I could learn all aspects of research from him — defining the right problem, setting up technically sound experiments, and writing the paper with a fresh angle. On the personal side, he has helped me endure physical and mental hardships by showing benevolence as well as providing motivation. It was such an honor being able to work with him.

Besides my advisor, I would like to thank my committee members — Prof. Cheng-Kok Koh, Prof. Kaushik Roy, and Prof. Vijay Raghunathan — for their valuable comments that helped my thesis to be in a better shape.

Special thanks to Dr. Swagath Venkataramani, who has been involved in all my research projects and always revised the paper to perfection. I admire his relentless enthusiasm for research and his positive attitude.

I thank all ISL colleagues whom I shared my on-campus life with. A wide range of discussions from research to culture and politics made my days in MSEE337 enjoyable. Thanks to Ashish, Shubham, Younghyun, Sanchari, Jacob, Manik, Sarada, Shrihari, Sourjya, Reena, Soumendu, Christin, Sanjay, Vinod, and Abinand. I wish all the best to everyone in their future endeavors.

Thanks to all my friends in the three main groups that I have actively participated in: ECE, PKTC, and SNUPU. May our paths cross again in the future.

Last but not the least, I'd like to thank my family: My parents and my sister for supporting me both financially and spiritually. Suyeon, my dear wife, for staying by my side through the endless Ph.D. study. Jiseob and Jiin, my lovely children, for being a better creation than any of my thesis chapters.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AxC	Approximate Computing
RTL	Register Transfer Level
FF	Flip-Flop
COT	Clock Overgating Target
COE	Clock Overgating Enable
MSB	Most Significant Bit
LSB	Least Significant Bit
VSX	Value Similarity eXtensions
IF	Instruction Fetch
SB	Subset Buffer
ARF	Access Redirection Function
FLOP	Floating-Point OPeration
ML	Machine Learning
DNN	Deep Neural Network
GPP	General Purpose Processor
VSX	Value Similarity eXtensions

ABSTRACT

Many modern workloads such as multimedia, recognition, mining, search, vision, *etc.* possess the characteristic of *intrinsic application resilience* — the ability to produce acceptable-quality outputs despite their underlying computations being performed in an approximate manner. *Approximate computing* has emerged as a paradigm that exploits intrinsic application resilience to design systems that produce outputs of acceptable quality with significant performance/energy improvement. The research community has proposed a range of approximate computing techniques spanning across circuits, architecture, and software over the last decade. Nevertheless, approximate computing is yet to be incorporated into mainstream HW/SW design processes largely due to the deviation from the conventional design flow and the lack of runtime approximation controllability by the user.

The primary objective of this thesis is to provide approximate computing techniques across different layers of abstraction that possess the two following characteristics: (i) They can be applied with minimal change to the conventional design flow, and (ii) the approximation is controllable at runtime by the user with minimal overhead. To this end, this thesis proposes three novel approximate computing techniques — *clock overgating* which targets HW design at the Register Transfer Level (RTL), *value similarity extensions* which enhance general-purpose processors with a set of microarchitectural and ISA extensions, and *data subsetting* which targets SW executing for commodity platforms.

Clock Overgating. The thesis first explores *clock overgating*, which extends the concept of clock gating — a conventional low-power technique that turns off the clock to a Flip-Flop (FF) when the value remains unchanged. In contrast to traditional clock gating, in clock overgating the clock signals to selected FFs in the circuit are gated even when the circuit functionality is sensitive to their state. This saves additional power in the clock tree, the gated FFs and in their downstream logic, while a quality loss occurs if the erroneous FF states propagate to the circuit outputs. This thesis develops a systematic methodology to identify an energy-efficient clock overgating configuration for any given circuit and quality constraint. Towards this end, three key strategies for efficiently pruning the large space of possible overgating configurations are proposed — significance-based overgating, grouping FFs into

overgating islands, and utilizing internal signals of the circuit as triggers for overgating. Across a suite of 6 machine learning accelerators, energy benefits of 1.36× on average are achieved at the cost of a very small (<0.5%) loss in classification accuracy.

Value Similarity Extensions. The thesis also explores *value similarity extensions*, a set of lightweight micro-architectural and ISA extensions for general-purpose processors that provide performance improvements for computations on data structures with value similarity. The key idea is that programs often contain repeated instructions that are performed on very similar inputs (*e.g.*, neighboring pixels within a homogeneous region of an image). In such cases, it may be possible to skip an instruction that operates on data similar to a previously executed instruction, and approximate the skipped instruction's result with the saved result of the previous one. The thesis provides three key strategies for realizing this approach — identifying potentially skippable instructions from user annotations in SW, obtaining similarity information for future load values from the data cache line currently being accessed, and a mechanism for saving & reusing results of potentially skippable instructions. As a further optimization, the thesis proposes to replace multiple loop iterations that produce similar results with a specialized instruction sequence. The proposed extensions are modeled on the gem5 architectural simulator, achieving speedup of 1.81× on average across 6 machine-learning benchmarks running on a microcontroller-class in-order processor.

Data Subsetting. Finally, the thesis explores a data-centric approach to approximate computing called *data subsetting* that shifts the focus of approximation from computations to data. The key idea is to restrict the application's data accesses to a subset of its elements so that the overall memory footprint becomes smaller. Constraining the accesses to lie within a smaller memory footprint renders the memory accesses more cache-friendly, thereby improving performance. This thesis presents a C++ data structure template called $SubsettableTensor$ — which embodies mechanisms to define an accessible subset of data and redirect accesses away from non-subset elements — for realizing data subsetting in SW. The proposed concept is evaluated on parallel SW implementations of 7 machine learning applications on a 48-core AMD Opteron server. Experimental results indicate that 1.33×–4.44× performance improvement can be achieved within a <0.5% loss in classification accuracy.

In summary, the proposed approximation techniques have shown significant efficiency improvements for various machine learning applications in circuits, architecture and SW, underscoring their promise as designer-friendly approaches to approximate computing.

1. INTRODUCTION

The development and wide-spread adoption of various computing devices that generate data, from mobile phones to Internet-of-Things (IoT) devices, have led to the emergence of applications that analyze and process large amounts of data to produce human-interpretable outputs. These emerging applications which include multimedia processing, machine learning, search and data analytics among others pervade into our everyday lives. While the evolution of these emerging applications demands more and more computing power to produce better-quality outputs within a reasonable time, the efficiency improvement due to technology scaling has been diminishing [1]. The challenge of this *efficiency gap* has forced both academia and industry to explore new directions for computing system performance and energy efficiency improvement.

Meanwhile, these prevalent emerging workloads possess a unique characteristic called *intrinsic application resilience* which enables them to produce an acceptable-quality output — a degradation in output quality which the user doesn't notice or can tolerate — despite their underlying computations being performed in an approximate manner. Intrinsic application resilience is associated with certain attributes of these applications, such as:

- A golden output doesn't exist, as is often the case in web search.
- Self-healing nature of iterative algorithms where an error introduced in one iteration eventually gets cancelled out during subsequent iterations.
- The fact that these applications — including Deep Neural Networks (DNNs) — are built to be robust to noisy and redundant real-world inputs.
- Limited capabilities of human perception, which allows some quality degradation without the user noticing it.

From the user's perspective, the expected result of these applications is not a unique golden numerical answer. Rather, the user expects a result of sufficient quality — in other words, an output that is "good-enough" — as depicted in Fig. 1.1. The fact that the users of these applications are satisfied with good-enough results, along with the intrinsic application

resilience possessed by these applications, has led to an emerging design paradigm called approximate computing.

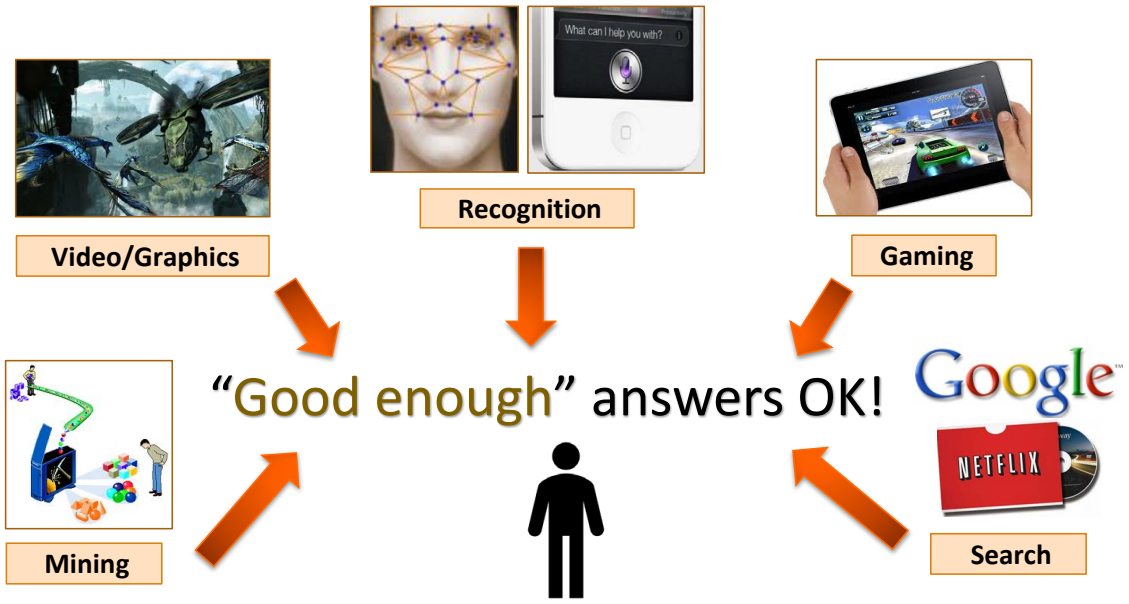


Figure 1.1. Modern applications producing good-enough answers

1.1 Approximate Computing

Approximate computing has risen as a new design paradigm that seeks the source of efficiency from designing SW/HW computing platforms in a way that they require just the minimum amount of effort (in terms of time, energy, or area) in order to produce "good enough" results. Approximate computing exploits intrinsic application resilience [2] by intentionally relaxing the notion of "exactness" or "correctness" on selected computations so that they are executed in a more efficient manner. Fig. 1.2 shows how approximate computing can be integrated into the traditional design process, which maps the functional specification of an application into software and the underlying hardware across multiple layers of design abstraction (*e.g.,* architecture, circuit and layout). While the traditional design process requires exact Boolean equivalence between the implementations in different layers of abstraction, approximate computing allows relaxation of the exactness requirement

so that a more efficient implementation is possible as long as the final output quality meets the given quality specification.

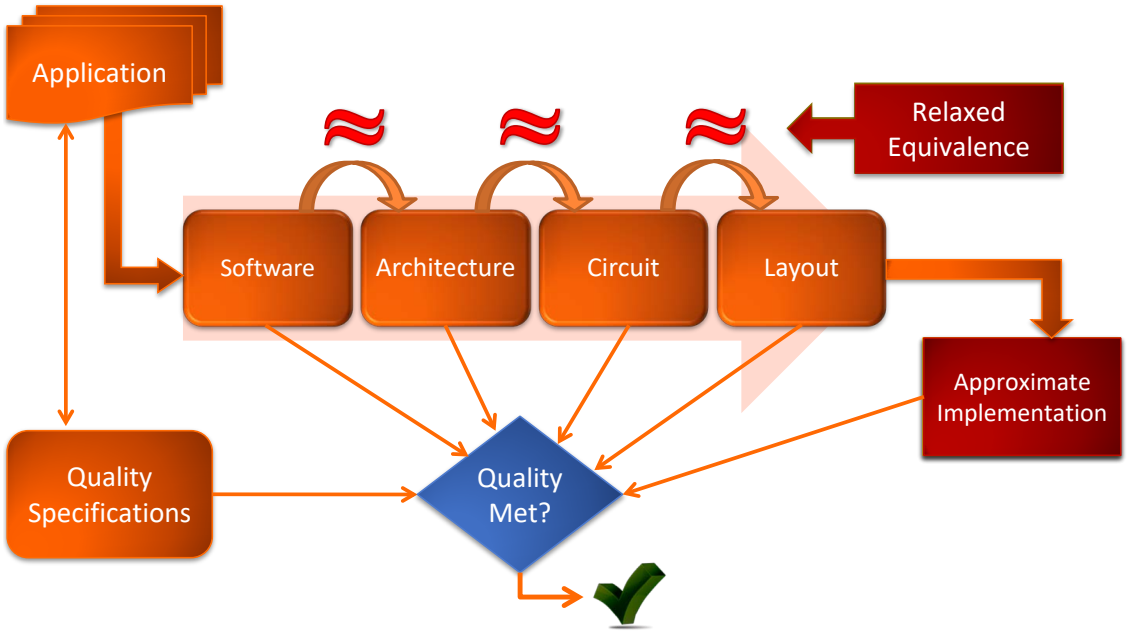


Figure 1.2. Approximate computing across the layers of abstraction [3]

Over the last decade, approximate computing has gained a significant interest among the research community due to the ample opportunity for efficiency improvement it provides. As a result, a large volume of approximate computing techniques spanning circuits, architecture, and software have been proposed. Despite the surge of approximation techniques, approximate computing is yet to be incorporated into mainstream HW/SW design practices. This thesis finds two main disadvantages of most existing approximate computing techniques that hinder them from being widely adopted:

Deviation from the conventional design flow. In case of HW design, most prior efforts have focused on approximation at the circuit or logic level of abstraction [4]–[15]. However, Register Transfer Level (RTL) is the most common layer of abstraction at which HW designs are specified and application-level output quality is addressed. In case of architecture design, many previous efforts have targeted specialized architectures such as domain-specific processors [16], GPUs [17], or custom accelerators [18], [19]. However, from the SW designer's point of view, utilizing specialized architectures itself requires a significant transformation

of the existing code with architectural knowledge. In case of SW design, many existing techniques require specialized HW and custom compilers making them difficult to adopt in practice [20]–[23].

Lack of runtime approximation controllability by the user. The user may often require the ability to take control over the degree or extent of approximation during runtime. However, most HW techniques hardwire approximations to the circuit implementation so that the degree of approximation is fixed [4]–[14]. SW techniques that rely on this type of underlying approximate HW also lack runtime controllability [20]–[23].

1.2 Thesis Contributions

The contribution of this thesis lies in exploring approximate computing techniques across different layers of abstraction that overcome the aforementioned limitations of prior works. Specifically, this thesis focuses on developing techniques that can be applied with minimal change to the conventional design flow, and are runtime configurable by the user with minimal overhead. This thesis suggests that re-thinking approximate computing from this perspective has a potential to take it closer to the mainstream. Towards this end, the thesis proposes three novel approximate computing techniques — *clock overgating*, which targets HW design at the Register Transfer Level (RTL), *value similarity extensions*, which enhances general-purpose processors with a set of microarchitectural and ISA extensions, and *data subsetting*, which targets SW design for commodity platforms.

1.2.1 Approximate Circuit Design using Clock Overgating

Clock overgating extends the concept of conventional clock gating into an approximate computing context. Clock gating is a low-power technique that turns off the clock to a Flip-Flop (FF) when the FF state remains unchanged. It is widely used in industry design practices and supported by most EDA tools which detect clock-enable conditions from the RTL code. While the traditional clock gating condition is confined in order to ensure the identical circuit functionality, clock overgating relaxes the gating condition to the point where the circuit functionality can be affected by the erroneous FF state. This saves switching

power in the clock tree, the target FF, and its downstream combinational logic at the cost of a possible quality loss in case the erroneous FF state propagates to the circuit output. The proposed technique enables approximation by modifying the clock gating conditions in the circuit's RTL description. This allows seamless integration into the existing design flow which typically starts from the RTL. In addition, approximation can be controlled in runtime by selecting the gating condition from multiple candidates. This thesis provides a systematic methodology to identify an energy-efficient clock overgating configuration for any given circuit and quality constraint is developed. The main challenge in identifying an optimal clock overgating configuration is that the number of candidate configurations to be evaluated grow exponentially with respect to the circuit size. In order to address this challenge, 3 key strategies for efficiently pruning the large space of possible overgating configurations are proposed — significance-based overgating, grouping FFs into overgating islands, and utilizing internal signals of the circuit as triggers for overgating. For a suite of 6 machine learning accelerators, energy benefits of 1.36× on average is achieved at the cost of a negligible (<0.5%) loss in classification accuracy.

1.2.2 Value Similarity Extensions for Approximate Computing in General-Purpose Processors

VSX (Value Similarity eXtensions for general-purpose processors) provides a set of microarchitectural and ISA extensions that exploits value similarity present in data structures for performance improvement. The key idea behind VSX is to dynamically detect and skip instructions that produce similar results to a previous one. In order to realize this approach, VSX must be able to identify potentially skippable instructions, dynamically detect an instruction result's similarity before execution, and approximate skipped instruction's result with a previous one. Towards this end, this thesis develops 3 key strategies for VSX — identifying potentially skippable instructions from user annotation in SW, obtaining similarity information for future load values from the data cache line currently being accessed, and providing a mechanism for saving & reusing results of potentially skippable instructions. As a further optimization, this thesis proposes iteration fast-forwarding, wherein multiple loop iterations producing similar results are substituted by a specialized sequence of instructions.

This thesis models VSX microarchitectural extensions on the gem5 simulator and runs 6 machine-learning benchmarks with real-word datasets on a low-end in-order processor platform. Speedups of 1.19×-3.84× (1.81× on average) are achieved across the benchmarks within a tight output quality constraint (<0.5%) while incurring a small HW area overhead (2.13%).

1.2.3 Data Subsetting: A Data-Centric Approach to Approximate Computing

The third body of work in this thesis shifts the focus of approximation from computations to data, and proposes a data-centric approach to approximate computing called *data subsetting*. The key idea is to modulate the application's data accesses in a manner that reduces off-chip traffic. Specifically, all accesses to data elements outside the selected subset are redirected to a subset element so that the overall memory footprint is reduced. Constraining the accesses to lie within a smaller memory footprint renders the memory accesses more cache-friendly, thereby improving performance. This approach allows approximations (access redirections) to be applied to the target SW code which eliminates the need for an approximate HW and a custom compiler. In addition, the degree of approximation can be dynamically reconfigured by adjusting the size and the redirection policy. This thesis presents a C++ data structure template called $SubsettableTensor$ which can be used for realizing data subsetting in SW design. $SubsettableTensor$ embodies mechanisms to define an accessible subset of data and redirect accesses to non-subset elements. As a further optimization, the thesis observes that data subsetting may cause some computations to become redundant and propose a mechanism for application software to identify and eliminate such computations. The proposed technique is evaluated by applying $SubsettableTensor$ to parallel SW implementations of 7 machine learning applications on a 48-core AMD Opteron server. Experimental results indicate that 1.33×–4.44× performance improvement can be achieved within a <0.5% loss in classification accuracy.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents a brief overview of the prior efforts in approximate computing classified into their target layers of abstraction. Chapter 3 describes the concept of clock overgating and the proposed systematic methodology for design approximate circuits using clock overgating. Chapter 4 describes how VSX exploits value similarity for performance improvement and the details of the micro-architectural and ISA extensions. Chapter 5 describes the concept of data subsetting and how it is applied to existing SW code for real-world applications. Finally, Chapter 6 concludes this thesis.

2. RELATED WORK

Approximating computing is a vibrant area of research and prior efforts have proposed approximation techniques spanning the different layers of the computing stack [1], [24], [25]. This chapter presents an overview of such efforts and highlight the key distinguishing features of the thesis.

2.1 Circuit Design Techniques for Approximate Computing

Research on approximate circuits began with manual designs of basic arithmetic components such as adders [4]–[8], [26]–[31] and multipliers [8], [9]. Approximate design techniques such as voltage overscaling [10], [11], [32]–[34], truncating carry chains [6], and input operand partitioning [7] were proposed in this context. Modifying the Boolean function realized by the circuit to reduce logic complexity was proposed as an approach to approximate circuit design in [12]. Subsequent efforts broadened this concept to the approximate design of arbitrary circuits using techniques such as redundancy propagation [13], path pruning [14], and don't care based simplification [15], substitution [35], logic isolation [36] among others. [37] provides error modeling and analysis on the approximate circuits. All of the above efforts operate at the logic level of abstraction, and target the design of approximate combinational circuits. How such approximate circuit blocks should be composed to form larger designs remains an open challenge. Chapter 3 proposes an approximate circuit design methodology at RTL, which is a higher level of abstraction targeting larger circuits with both combinational and sequential logic.

2.2 Architectures for Approximate Computing

At the architecture level, research efforts build approximate architectures both in the context of application-specific designs and programmable processors. Means of approximation came from various sources as operating at sub-threshold voltage [38], stochastic processing [39], underlying approximate HW [20], [22], [40]–[60] or quality-configurable vector processing unit [16] where compiler support is provided for mapping variables/operations in

SW to their corresponding approximate HW storage/computation components. While the majority of existing techniques target only the execution pipeline stage, Chapter 4 proposes a technique which increases the efficiency of instruction fetch and decode as well.

2.3 Software Techniques for Approximate Computing

The majority of SW works in approximate computing have focused on providing programming language/compiler support for approximating internal computations/functions, that are categorized as *compute-centric approaches*. One branch of compute-centric approach maps computations/functions onto approximate HW[20]–[22], [41], [61]–[66], neural network[23], user-defined approximate loops/functions[67], approximate kernels for commonly used data parallel computation patterns[68], piecewise polynomial approximation of the target function[69], or a computational model which increases the output quality over time[70]. Another branch of compute-centric approach judiciously skips loop iterations[71], [72], atomic instructions[73], tasks[74] or critical sections[75]. In contrast to the aforementioned compute-centric approaches, Chapter 5 proposes a data-centric approach to AxC. As opposed to approximating computations, Chapter 5 targets modulating data structure accesses in a manner that reduces memory bandwidth, thereby speeding-up applications executed on memory-bound platforms.

Another category of approximate SW work targets multi-threaded programs and achieve approximation by relaxing dependencies or synchronization constraints. [76] immediately terminates the current phase on all processors when the majority of them are in idle condition. [77] forces the threads in a SIMD execution to take the same control path or read from the same memory block. [75], [78] proposes a parallelizing compiler and runtime which provides control knobs for relaxing inter-loop dependencies and synchronizations. While these works are orthogonal to the approach proposed in Chapter 5, the proposed approach for mitigating memory bottleneck is especially effective in multi-threaded environments, thus can be used in complementary to each other.

3. APPROXIMATE CIRCUIT DESIGN USING CLOCK OVERGATING

3.1 Introduction

Many prevalent and emerging application domains such as multimedia, recognition, mining, data analytics, search and vision among others, possess the characteristic of *intrinsic application resilience*, which enables them to produce outputs of acceptable quality, despite approximations to some of their computations. Intrinsic application resilience arises from several factors: algorithms are designed to be robust to noise, input data contains redundancies, applications only need to produce an acceptable output rather than a unique golden result, *etc.* [24], [79]. Approximate computing leverages intrinsic application resilience and relaxes the need for strict correctness in the execution of computations to improve energy and/or performance.

Over the last decade, several research efforts have developed techniques for approximate computing in both software [24], [71] and hardware [4]–[15], [80]–[83]. Approximate hardware may be realized by operating circuits under overscaled conditions [10], [11] (*e.g.*, lowering the voltage to a point where timing errors occur), or by designing hardware that contains fewer transistors or gates but deviates to a limited extent from the specification [4]–[9], [12]–[15], [80], [81]. A vast majority of approximate hardware design techniques are at the circuit or logic levels of abstraction. It is desirable to explore approximate design at the higher levels of abstraction, since from the designer's point of view, quality is most naturally defined and evaluated at the application level. For example, it is much more natural to talk about classification accuracy or image quality than to specify accuracy or error constraints at the outputs of individual adders or multipliers within a circuit.

The Register Transfer Level (RTL) remains the most common level of abstraction at which hardware designs are specified in the industry. In addition to inherent advantages in terms of faster simulation and analysis times, RTL descriptions contain higher-level semantic information (data types, bit positions, operators, *etc.*) that can be leveraged in the process of approximation.

Very few research efforts have previously investigated approximate computing at the RTL. Notably, Axilog [83] proposes Verilog HDL extensions that designers can use to specify which signals and operations in the RTL code can be subject to approximations. In contrast, this chapter's objective is to develop a new approximate design technique, and hence Axilog [83] is complementary to this chapter's proposal. Another effort, ABACUS [82], generates approximate designs from behavioral/RTL descriptions by applying pre-specified approximation transformations to individual operations or operands. A key drawback of this approach is that the approximations are fixed at design time and hardwired into the circuit implementation. This is an issue in practice, since the same hardware may be used to execute computations that can tolerate different levels of approximation, or the application context may dictate the use of different accuracies for the same computation. These considerations strongly suggest that the approximations employed should be reconfigurable at runtime.

This chapter proposes a new approximation technique called *clock overgating* for the design of approximate hardware. Clock gating is one of the most widely adopted low-power techniques, in which clock signals to sequential elements (flip-flops or latches) in the circuit are suppressed to reduce power, provided that doing so preserves the exact functionality of the circuit. This chapter extends this concept to propose clock overgating, where the clock signal to a Flip-Flop (FF) is gated *even when doing so may affect the circuit outputs*. Clock overgating may result in incorrect outputs when an FF is supposed to change state but incorrectly retains its previous state, and this error propagates to the circuit outputs. In return, switching power is saved in the clock tree, the FF itself, and the fanout logic cone of the FF. Note that these power savings are over-and-above the savings due to conventional clock gating and other low-power techniques.

Clock overgating has the following desirable properties: (i) It is easily reconfigurable, *i.e.*, the clock overgating signal to each FF can be modulated dynamically in a fine-grained manner, (ii) it preserves the structure of the circuit and is hence minimally-intrusive, and (iii) it leverages the wide support for clock gating that is present in commercial EDA tools and design flows.

Given the RTL description of a circuit, an input testbench and an output quality constraint, this chapter proposes a systematic methodology to identify where (in which FFs)

and when (in which clock cycles) to perform overgating. The search space for overgating, defined by all possible FFs and all possible execution cycles, is extremely large (*e.g.*, 2^{10000} for a circuit with 100 FFs that operates for 100 cycles), and is even more challenging in cases when designs take a variable number of cycles to complete execution. Rather than explicitly search through this prohibitively large space, the proposed methodology utilizes internal signals from the circuit to trigger clock overgating. Doing so restricts the search space, and also has the added benefit of incurring minimal overhead in the logic that realizes the overgating conditions. Further, FFs in a circuit are grouped into clock overgating islands based on their location in the circuit and how they impact the overall application output. FFs in each overgating island are constrained to have the same overgating condition, greatly reducing the search space without significantly affecting the energy savings from overgating. Then a gradient descent search is performed to identify the set of clock overgating island–trigger signal combinations that maximize the improvement in energy for the specified quality constraint.

In summary, the key contributions of this work are as follows:

- This chapter proposes clock overgating, a new technique to design approximate circuits at the RTL, in which sequential elements in the circuit are clock gated even when doing so may affect the circuit outputs.
- This chapter develops a systematic methodology to determine the clock overgating conditions for sequential elements in any given circuit. The proposed methodology groups sequential elements into overgating islands and uses internal circuit signals as overgating triggers to efficiently prune the space of possible overgating configurations.
- This chapter applies clock overgating to develop approximate versions of accelerators for 6 machine learning applications, and demonstrate 1.36× average improvement in energy for <0.5% loss in quality at the application-level.

3.2 Comparison to Related Work

Section 2.1 presented various research efforts on approximate circuit design and emphasized their limitations where the approximation target is restricted to combinational circuits

and performs approximation at the logic level of abstraction. In contrast to the aforementioned efforts, this chapter focuses on a higher level of abstraction, *viz.* RTL, where very few efforts have explored approximate computing. Axilog [83] provides Verilog HDL extensions, which can be used to specify and identify portions of the design that are safe to approximate. Axilog is complementary to this work as this work's objective is to develop new approximate design techniques that can be applied to the identified parts. ABACUS [82] applies predefined approximate transformations (*e.g.*, scaling bitwidth, strength reduction) to operators in the RTL description. However, a key limitation of [82] is that the approximations are hardwired into the circuit implementation, and cannot be changed at runtime. In practice, hardware needs to be re-used in different application contexts, operate on different inputs, or execute different operations within an application; all these scenarios require the ability to modulate the degree of approximation at runtime. Clock overgating is inherently quality configurable, since one can enable overgating in different subsets of FFs to realize distinct energy *vs.* quality tradeoffs. Furthermore, it has other desirable attributes such as preserving the overall structure of the circuit and can be easily integrated into existing design flows. Finally, recent work has explored High-Level Synthesis (HLS) of approximate circuits through precision scaling [81]. Notwithstanding significant advances in HLS, it is noted that hardware is often still designed at the RTL in the industry, hence this chapter focuses on the complementary problem of approximating any given RTL design in a quality-configurable manner.

3.3 Background

3.3.1 Power Dissipation in Sequential Circuits

The average power dissipated in a sequential circuit can be broken down into is major components as:

$$P_{average} = P_{leakage} + P_{dynamic} \tag{3.1}$$

$P_{leakage}$ mainly comes from sub-threshold currents and they contribute to the average power dissipation even when there are no switching activities. The portion of $P_{leakage}$ with

respect to $P_{average}$ increases as the transistor size becomes smaller. On the other hand, $P_{dynamic}$ is caused by switching activities of each node, where the load capacitance needs to be charged. $P_{dynamic}$ can be represented as:

$$P_{dynamic} = \alpha \cdot C_L \cdot {V_{DD}}^2 \cdot f \tag{3.2}$$

Where α being the fraction of the circuit that is switching switching, $\cdot C_L$ being the load capacitance, V_{DD} being the supply voltage, and f being the operation frequency. The part of a circuit where $P_{dynamic}$ becomes dominant is the clock tree, where the input clock is constantly switching while the circuit is powered on.

3.3.2 Clock Gating

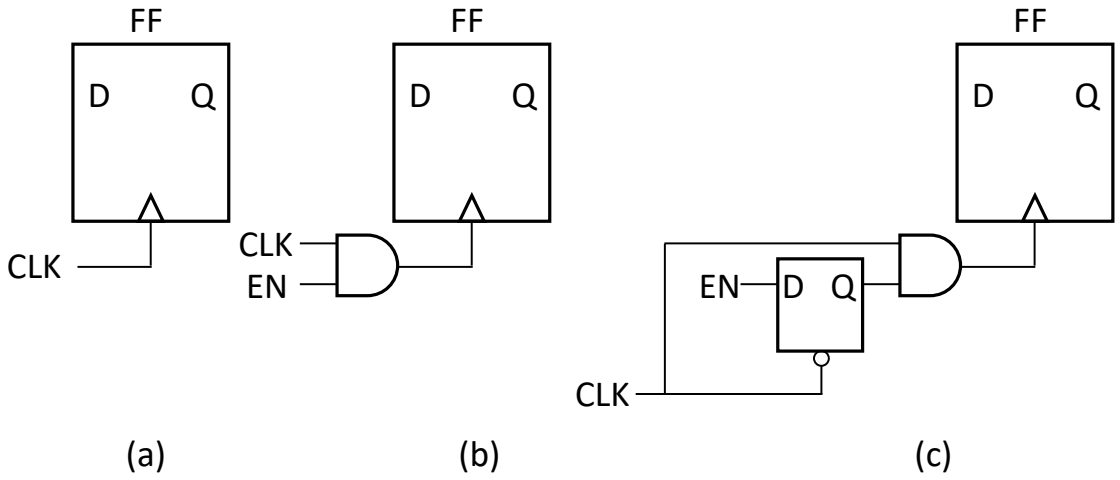


Figure 3.1. Different types of clock gating logic

As mentioned in Section 3.3.1, clock power occupies a significant fraction of the total dynamic power dissipation since the clock is fed to every circuit block where sequential logic elements (*e.g.,* flip-flops) are present. Clock gating is a wide-spread lower-power techinique which targets clock power. The key concept of clock gating is to turn off the clock to a Flip-Flop (FF) when the FF state does not change. Gating conditions can be explicitly specified by the user (explicit load-enable condition), or can be derived by comparing the next cycle's input to the current state (XOR-based approach). Fig. 3.1 shows different types

of clock gating logics which are widely used. In contrast to Fig. 3.1(a) where the CLK signal is directly fed to the target FF, Fig. 3.1(b) gates CLK with an EN signal and an AND gate so that the input clock to the FF doesn't switch when EN is low. Fig. 3.1(c) is an advanced type of clock gating logic (latch-based) where glitches to EN does not affect the functionality of the target FF. Clock overgating technique presented in Chapter 3 extends the conventional duration of EN so that the clock remains gated for a longer amount of time while the FF state becomes erroneous. The selection of EN for each FF is judiciously made through a systematic methodology so that each FF is gated longer while minimizing the effect on the final output.

3.4 Design Approach

Given a hardware design described at the RTL and a quality constraint specified at its outputs, this chapter's objective is to design an approximate version through clock overgating that is as energy-efficient as possible, while meeting the specified quality constraints. This section describes the basic concepts behind clock overgating, the challenges involved in applying it to the design of approximate circuits, and this chapter's approach to addressing these challenges.

3.4.1 Clock Overgating: Concept

Clock gating is a popular low-power design technique that is widely used to reduce dynamic power dissipation. In clock gating, additional logic is embedded in the clock tree of the circuit, which suppresses the clock signal transitions from reaching one or more sequential elements (FFs or latches) in the circuit under certain conditions. Since the clock signal is suppressed, the target FFs cannot switch states and therefore switching power is saved in the clock tree leading to the FFs, the FFs themselves and their fanout logic cones. To preserve functionality, clock gating is performed only when an FF's state is guaranteed not to switch, or if it can be proven that the circuit's outputs are insensitive to the FF's inputs in the gated execution cycles.

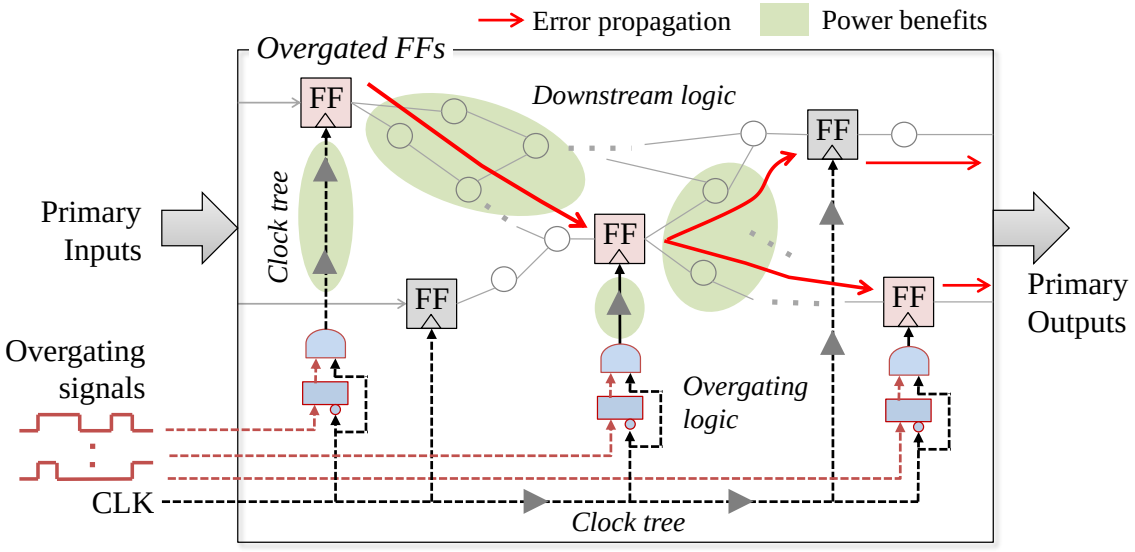


Figure 3.2. Clock overgating concept

This chapter proposes clock overgating, whose concept is illustrated in Fig. 3.2. In clock overgating, selected FFs in the circuit are gated even when doing so may result in an incorrect circuit output. In other words, local errors are introduced in the circuit due to incorrect FF states, and they may propagate to the circuit output, potentially causing the output quality to degrade. In return, since FFs are being gated for additional execution cycles beyond conventional clock gating, it results in improved dynamic power and energy. Thus, clock overgating presents designers with a means to tradeoff energy *vs.* quality. The key question that arises, which will be discussed next, is how to utilize clock overgating to achieve the best energy benefit for a given output quality?

3.4.2 Clock Overgating: Design Space

Identifying a *clock overgating configuration* involves defining: (i) which FFs in the circuit should be overgated, and (ii) when (during which clock cycles) the selected FFs should be overgated, or in other words, what conditions should be used to trigger overgating for the selected FFs. This chapter refers to the FFs that are candidates for overgating as Clock Overgating Targets (COTs), and the signals used to trigger clock overgating in them as Clock Overgating Enables (COEs). Fig. 3.3 shows the design space for clock overgating.

If M is the number of FFs in a circuit, which operates for N cycles, then each FF can be either gated or active in each cycle, leading to a total of 2^{M*N} overgating configurations. Clearly, a brute force search of all possible overgating configurations is infeasible for practical designs. Therefore, efficient heuristics are key to exploring this very large design space and to identifying the clock overgating configuration that is most energy-efficient for a given output quality constraint (QC).

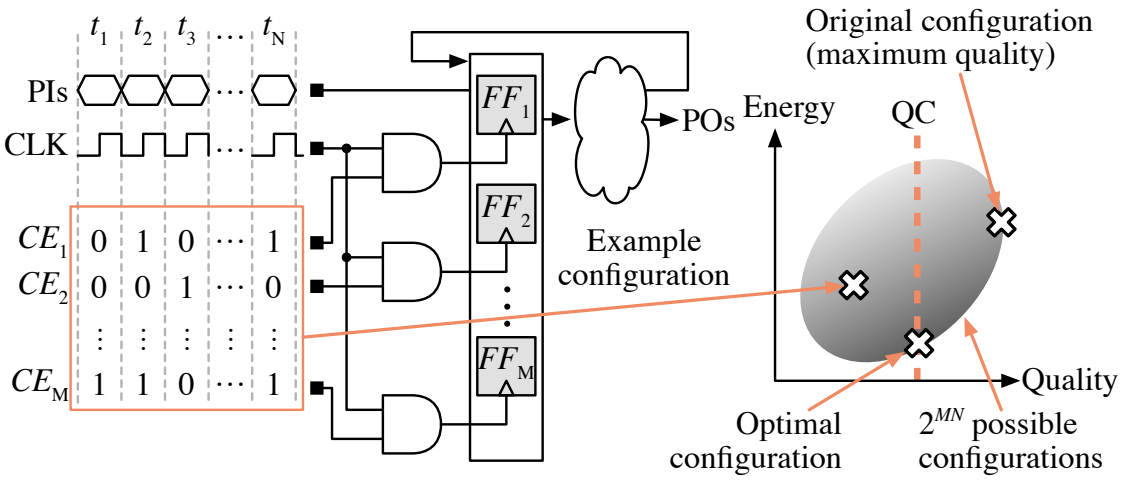


Figure 3.3. Design space of clock overgating configurations

To address the above challenge, this chapter adopts 3 strategies that leverage the semantic information available at the RTL to significantly reduce the number of COTs as well as the number of possible COEs for each COT. Although these strategies do not guarantee optimality, *i.e.*, they may result in a suboptimal overgating configuration, this chapter's experimental results demonstrate that they work well in practice and yield significant improvements in energy even for very tight quality constraints.

Significance constrained overgating: In general, each FF in the circuit can be regarded as a COT and overgated independently. However, in practice, multiple FFs are semantically grouped into registers that together store/represent multi-bit data. This chapter extracts this information from the RTL description of the circuit and exploits the bit significance of individual FFs in their respective registers to constrain the sequence in which they are overgated. In typical circuits, errors in the LSBs of registers impact the output quality by

an exponentially smaller amount relative to the MSBs. Further, the LSBs typically have the most switching activity and fan out to larger cones of logic; hence, they can lead to higher energy savings when overgated. Therefore, identification of COEs for FFs in a given register is proceeded in the order of their bit significance, *i.e.*, COEs for the LSBs are identified first before considering the MSBs for overgating. In summary, the first strategy prunes the search space by associating FFs with registers specified at the RTL, and constraining the manner in which they are overgated based on bit-significance.

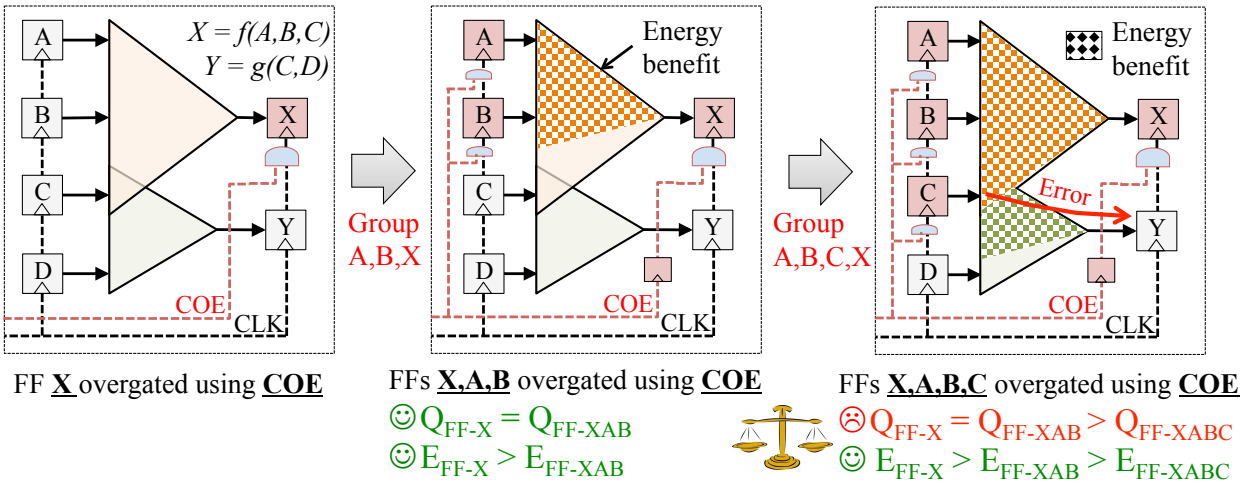


Figure 3.4. Functional grouping of FFs

Functional grouping of FFs: In this strategy, this chapter reduces the number of COTs by grouping multiple FFs into overgating islands, and overgating all FFs in an island together. This is achieved by leveraging the functional relationship between FFs that can be easily identified at the RTL. Fig. 3.4 illustrates the opportunity for FF grouping. Consider a scenario where clock overgating in FF X is triggered by signal COE. This enables energy savings in X and in its downstream logic. Now, as shown in Fig. 3.4, X is functionally related to FFs A, B and C *i.e.* X changes state only in response to a change in A, B or C. Also, note that FFs A and B exclusively fan out to X, whereas C also fans out to other FFs in the circuit. In this case, the key observation is that when COE is used to clock overgate FFs A and B in the cycle before X is overgated, it results in no additional quality loss at the circuit outputs, as any errors introduced at A and B propagate only to X, which itself

is overgated in the next cycle. However, this results in additional energy savings in the FFs A and B, and in the combinational logic connecting FFs A and B with X. Therefore, this chapter groups FFs X, A and B into a single overgating island, and uses the same signal (COE) to overgate all of them. Note that when grouped, the overgating signal to X is delayed by one clock cycle, as the values in A and B propagate to X only after a cycle.

The above strategy can be extended to include an FF in a group even when some of its fanouts lie outside the group. In this case, grouping may result in additional quality degradation, as errors introduced in the FFs due to overgating can propagate to the circuit outputs through its fanouts that lie outside the island. In the example shown in Fig. 3.4, if C is grouped along with X, A and B, the error introduced in C impacts its fanouts other than X, leading to an additional degradation in quality. This additional quality degradation needs to be compared against the net energy benefits to decide if it is favorable to add an FF to an overgating island. A detailed description of how FFs are grouped into overgating islands is presented in Section 3.5.

Using internal signals as COEs: The final strategy addresses the large space of possible COE candidates that could be used to trigger overgating in each COT. Since logic to generate COEs needs to be added to the circuit, a key constraint in picking COEs is that they should incur low overhead. To reduce the search space, while also minimizing the energy overheads in the logic that generates COEs, this chapter proposes to use internal signals already present within the circuit as COEs. In addition, to ensure that the timing constraints on the gating signal are satisfied, and that no glitches are introduced on the gated clock signals, this chapter further restricts COEs to be the outputs of FFs in the circuit and their complements. While this restriction greatly prunes the design space, this chapter's experiments suggest that it still enables a rich and favorable energy-quality tradeoff. This chapter suggests that in practice, circuits contain control states such as address counters, FSM state variables *etc.* that naturally predict which registers in the circuit are active and likely to affect the output the most.

In summary, this chapter proposes clock overgating, an RTL approximation technique in which clock signals to FFs in a circuit are gated even when the circuit outputs may be

affected, thereby resulting in a tradeoff between energy and output quality. This chapter also proposes three key strategies to prune the space of possible overgating configurations, enabling clock overgating to be applied to any given input circuit while meeting the desired quality constraint.

3.5 Design Methodology

This section presents a systematic methodology for clock overgating that realizes the design approach described in Section 3.4.

3.5.1 Approximate Design using Clock Overgating

Algorithm 1 Approximate design using clock overgating

Input: Original circuit Ckt_{orig},
Application dataset $AppData$, Quality constraint QC
Output: Approximate circuit Ckt_{app}

1: COT_{List} = form_COTs(Ckt_{orig})
2: $COT_{subList}$ = select COTs in LSB position from COT_{List}
3: COE_{List} = FFs and their complements in Ckt_{orig}
4: Ckt_{opt} = Ckt_{orig} with COE=0 $\forall$ COTs in COT_{List}
5: **do**
6: Ckt_{app} = Ckt_{opt}; FOM_{opt} = 0
7: COT-COE_{List}={<T,E> | $T \in COT_{subList}$, $E \in COE_{List}$}
8: **for** each <COT,COE> in COT-COE_{List} **do**
9: Ckt_{tmp}=assign COE as ovg. signal for COT in Ckt_{app}
10: ΔE = E_{orig} - get_energy(Ckt_{tmp},$AppData$)
11: ΔQ = Q_{orig} - get_quality(Ckt_{tmp},$AppData$)
12: FOM = $\Delta E/\Delta Q$
13: **if** (ΔE>0 **and** ΔQ<QC **and** FOM>FOM_{opt}) **then**
14: COT_{opt} = COT; COE_{opt} = COE
15: Ckt_{opt} = Ckt_{tmp}; FOM_{opt} = FOM
16: **end if**
17: **end for**
18: remove COT_{opt} from $COT_{subList}$
19: $COT_{subList}$ = add COT containing next significant bit of COT_{opt} from COT_{List}
20: **while** ($Ckt_{app} \neq Ckt_{opt}$)
21: **return** Ckt_{app}

Algorithm 1 shows the pseudocode of the proposed methodology. Given the original circuit (Ckt_{orig}), an input dataset ($AppData$), and a constraint on the output quality (QC), the algorithm produces an energy-efficient approximate version (Ckt_{app}) that is clock overgated and satisfies the specified quality constraint. First, a list of possible clock overgating targets (COT_{List}) and clock overgating enables (COE_{List}) are formed by employing the different design space pruning strategies described in Section 3.4 (lines 1-3). To this end, the FF grouping strategy is first used to form the COT_{List} (line 1). The details of this process are explained in Algorithm 2. Next, the significance constrained overgating strategy is used to further reduce the list of COTs (resulting in $COT_{subList}$). In this case, only the COTs that are in the least significant bit positions in their corresponding registers are chosen from COT_{List}, and added to $COT_{subList}$ (line 2). Next, since only FF outputs and their complements are considered as possible clock gating enables, COE_{List} is initialized with all FFs and their complements in the circuit (line 3).

Despite significantly reducing the design space using the various strategies, trying out all possible COT-COE combinations becomes computationally expensive for larger circuits. Therefore, this chapter adopts a gradient descent approach (lines 5-20), where in each iteration the COT-COE pair that yields the best energy *vs.* quality tradeoff without violating QC is selected. To this end, this chapter first initializes the COE for each COT to logic 0, which corresponds to no overgating (line 4). Then, this chapter forms a list of all possible COT-COE pairs ($COT\text{-}COE_{List}$) by pairing each COT in $COT_{subList}$ with each COE in COE_{List} (line 7). Next, for each COT-COE pair this chapter forms a candidate overgated circuit by assigning the COE under consideration as the overgating signal for the COT (line 9). Subsequently, this chapter evaluates the energy benefits (line 10) and the corresponding quality loss (line 11). Note that COT-COE pairs that cause unacceptable quality degradation are captured during quality evaluation and filtered out from further $COT\text{-}COE_{List}$ formations. This chapter then computes a figure of merit (FOM) for each COT-COE pair, which is given by the ratio of energy saved to quality lost (line 12). This chapter identifies the COT-COE pair that has the best FOM (lines 13-16) and updates the approximate circuit by "committing" the COE to the COT (line 15). The committed COT is removed from $COT_{subList}$ (line

18), and the COT containing the next significant bit position is added to $COT_{subList}$ (line 19). This process is repeated until no COT-COE pair yields further improvements in energy without violating QC (line 20), at which point the algorithm terminates and returns the approximate circuit (line 21).

3.5.2 Identifying Clock Overgating Candidates

Algorithm 2 Identify overgating candidates

Input: Original circuit Ckt, Application dataset $AppData$
Output: Clock overgating candidate list: COT_{List}
1: COT_{List} = FFs in Ckt
2: G = create_FF_connection_graph(Ckt)
3: W = get_RTL_simulation_waveform(Ckt,$AppData$)
4: $TSC(FF_1,FF_2)$: temporal correlation between FF_1&FF_2 in W
5: **do**
6: **for** each COT in COT_{List} **do**
7: **for** each COT_{In} in fan-in(COT,G) **do**
8: **for** each COT_{In-out} in fan-out(COT_{In},G) **do**
9: $TSC_{tmp} = TSC(COT_{In},COT_{In-out})$
10: ΔE_{local} += TSC_{tmp} × fan-out-size(COT_{In},COT_{In-out})
11: **if** ($COT_{In-out} \neq COT$) **then** ΔQ_{local} += TSC_{tmp}
12: **end for**
13: **if** ($\Delta E_{local}/\Delta Q_{local} > \eta$) **then**
14: COT_{List} = merge(COT,COT_{In})
15: **end if**
16: **end for**
17: **end for**
18: **while** (COT_{List} modified?)
19: **return** COT_{List}

Algorithm 2 presents the methodology used to obtain the list of clock overgating candidates in line 1 of Algorithm 1. Algorithm 2 essentially implements the FF grouping strategy presented in Section 3.4. First, COT_{List} is initialized to contain all the FFs in Ckt (line 1). Then COT_{List} is reduced by grouping FFs whose switching activities are highly correlated (lines 2-18). To this end, a FF connection graph (G), whose vertices represent the FFs and whose edges represent a combinational path between two FFs, is constructed from the RTL code (line 2). Then, the circuit RTL is simulated with the given dataset, and the switching

activity of each FF is recorded (line 3). Subsequently, the degree of 1-cycle temporal switching correlation (TSC) between FF pairs that have an edge connecting them in G is computed (line 4). This chapter denotes two FFs to be temporally correlated if they are likely to switch in successive cycles. Next, for each COT and its fan-in (COT_{In}), this chapter computes the local energy savings expected in all the fan-outs of COT_{In} if it is overgated along with COT (line 10). Similarly, this chapter evaluates the local errors expected in the fan-outs of COT_{In} that are not contained within COT, assuming that COT_{In} is overgated along with COT (line 11). This chapter merges COT_{In} with COT if the ratio of the local energy savings to the local error induced is greater than a threshold η (lines 13-15). In this chapter's implementation, the value of η is determined empirically as discussed in Section 3.7.3. This process is repeated until no pair of candidates in COT_{List} can be merged (line 18), and the final COT_{List} is produced (line 19).

3.6 Experimental Setup

This section describes the experimental setup used to evaluate clock overgating.

3.6.1 Benchmarks

The benchmark circuits consist of algorithm-specific hardware accelerators for 6 popular machine learning applications listed in Table 5.1. Classification accuracy, or the fraction of application inputs correctly classified, was used as the measure of output quality for all the benchmarks. Separate datasets were used for training and testing/validation, and the same testing dataset was used to evaluate the original and approximate circuits.

3.6.2 Quality and Power Evaluation

The design methodology and heuristics described in Section 3.5 were implemented as a custom script that executes the following design flow. First, a custom RTL parser extracts necessary information from the input RTL code and produces an approximate version. The output quality (classification accuracy) of the approximate circuit is obtained by RTL simulation with the testing dataset using Mentor Graphics ModelSim. RTL simulation also

Table 3.1. Benchmark applications

Algorithm	Application	Dataset	FFs/Gates
GLVQ	Eye detection (EYE)	Image set from NEC labs	183 / 3100
KNN	Optical digit classification (OPT)	OCR digits	344 / 4128
	Webpage classification (W5A)	LIBSVM	422 / 7348
SVM	Text classification (TEXT)	Reuters	480 / 27760
ANN	Digit classification (DIGT)	MNIST	398 / 29833
	Face detection (FACE)	YUV faces	394 / 31428

produces the switching activity information that is used for power estimation. Power is estimated by synthesizing the approximate circuit to IBM 45nm technology using Synopsys Design Compiler, and estimating the circuit power including the clock tree power using Synopsys PrimeTime. This chapter notes that the most aggressive optimization options available by Synopsys Design Compiler (conventional clock gating, gate sizing, use of low leakage cells, *etc.*) were applied to create well-optimized baselines for comparison.

3.7 Experimental Results

This section presents the results of various experiments that demonstrate the benefits of clock overgating and underscores the effectiveness of the proposed strategies in exploring the large design space of overgating configurations.

3.7.1 Energy-Quality Tradeoff

Fig. 3.5 shows the energy reduction obtained using clock overgating for various output quality constraints. Note that output quality constraints are expressed in terms of classification accuracy degradation with reference to the baseline, and the energy consumption of each approximate circuit is normalized to the energy consumption of its baseline implementation. As shown in Fig. 3.5, clock overgating provides significant energy benefits between 1.07×–1.80× (1.36× on average) at the application level with virtually no loss (<0.5%) in classification accuracy. When the quality constraints are relaxed to <2% and <5%, the

energy benefits increase to 1.10×–1.80× (1.43× on average) and 1.12×–1.80× (1.50× on average), respectively.

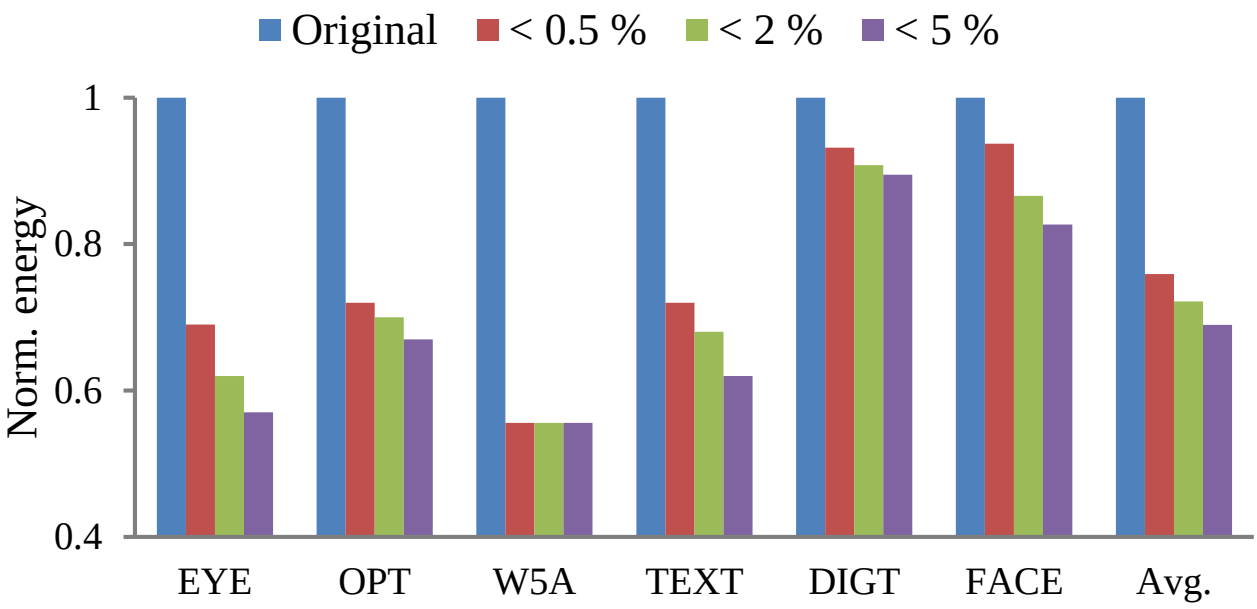


Figure 3.5. Energy improvements for various quality constraints

The graph shown in Fig. 3.6 breaks down the energy from Fig. 3.5 into 3 key components: (i) leakage (static) power, (ii) dynamic power in the registers and the clock tree, and (iii) dynamic power in the combinational logic. This section only presents the <5% quality constraint case here, but the observations hold for other cases as well. Again, all the energy components are normalized to the baseline implementation. First, this chapter finds that the overhead due to the increase in leakage power is negligible (0.21% on average). This suggests that the overhead for clock overgating was kept minimal (cell area increased by 0.28% on average) by using the internal signals as COEs. Next, it is observed that clock overgating reduces the power dissipated in the clock tree, the FFs' internal power, as well as the dynamic power consumed by the combinational logic. A general trend shown in Fig. 3.6 is that the power improvements are more pronounced in the combinational logic than in the clock tree and the FFs. This is due to the fact that only a small fraction (16.9% on average) of the total FFs are clock overgated in the design, and further, these overgated FFs are dominated by LSBs. The parts of the combinational logic that process LSBs typically

exhibit higher switching activity and therefore overgating the corresponding FFs results in considerable dynamic power savings without a significant impact on output quality.

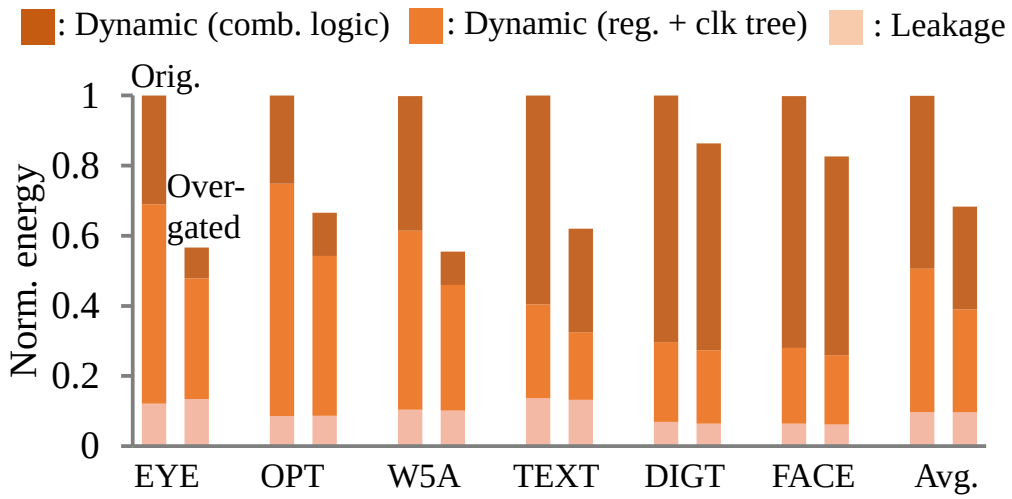


Figure 3.6. Energy breakdown analysis

3.7.2 Comparison with Precision Scaling

The graph shown in Fig. 3.7 compares the energy benefits achieved by clock overgating and precision scaling for various target quality constraints. In the case of precision scaling, different numbers of LSBs were clock gated in all data path registers, throughout the evaluation of the circuit. It can be clearly seen that clock overgating outperforms precision scaling in terms of energy benefits; for example, 1.36× *vs.* 1.19× on average for a quality constraint of <0.5%. This is due to the fact that clock overgating allows for a more fine-grained control over which FFs should be gated in which execution cycles.

3.7.3 Effectiveness of FF Grouping

As described in Section 3.4, this chapter groups multiple FFs into overgating islands, which significantly cuts down the search space of overgating configurations. However, this renders clock overgating to be more coarse grained, potentially impacting the energy efficiency of the overgated circuit. Fig. 3.8 quantifies this tradeoff for 2 benchmarks by varying the number of COTs into which the FFs of the circuit are grouped, and observing the runtime

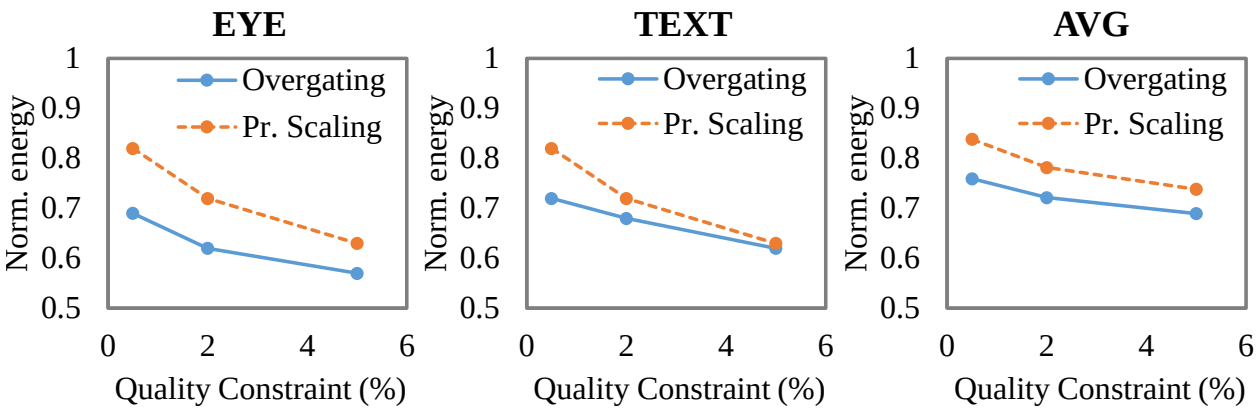


Figure 3.7. Energy benefits using clock overgating *vs.* precision scaling

of the methodology as well as the energy efficiency of the resulting overgated circuit. Note that, in Fig. 3.8, energy is normalized to that of the baseline circuit. A quality constraint of <5% classification accuracy was used in these experiments; however, the conclusions hold for other quality constraints as well. It is observed that the runtime increases with an increase in the number of COTs. In the case of energy, the benefits of clock overgating are very small when FFs are aggressively grouped. The energy efficiency of the overgated circuit improves drastically once the number of COTs crosses a threshold. However, beyond a point, the benefits saturate and very fine grained overgating does not yield additional improvements. Thus, if carried out to the right extent, FF grouping can achieve significant improvements in overall runtime, without a significant loss in the energy benefits. For instance, in the case of EYE, when its 183 FFs are grouped to form 129 COTs (using η=3.5 in Algorithm 2), the runtime reduces by 5.8× at the cost of a mere 1.8% decrease in energy savings. The result of this experiment shows that η values of 3.2–3.5 work well for all circuits. Note that by combining the strategies described in Section 3.4, the total runtime was reduced substantially to <4 hours for a circuit size of ∼30K gates. While this chapter argues that this is a reasonable runtime for this chapter's proof-of-concept implementation, it can be considerably reduced by utilizing static techniques (*vs.* simulation) for switching activity estimation, and parallelizing across multi-cores (for example, the loop over COT-COE combinations in Algorithm 1 can be parallelized) to ensure scalability for larger circuits.

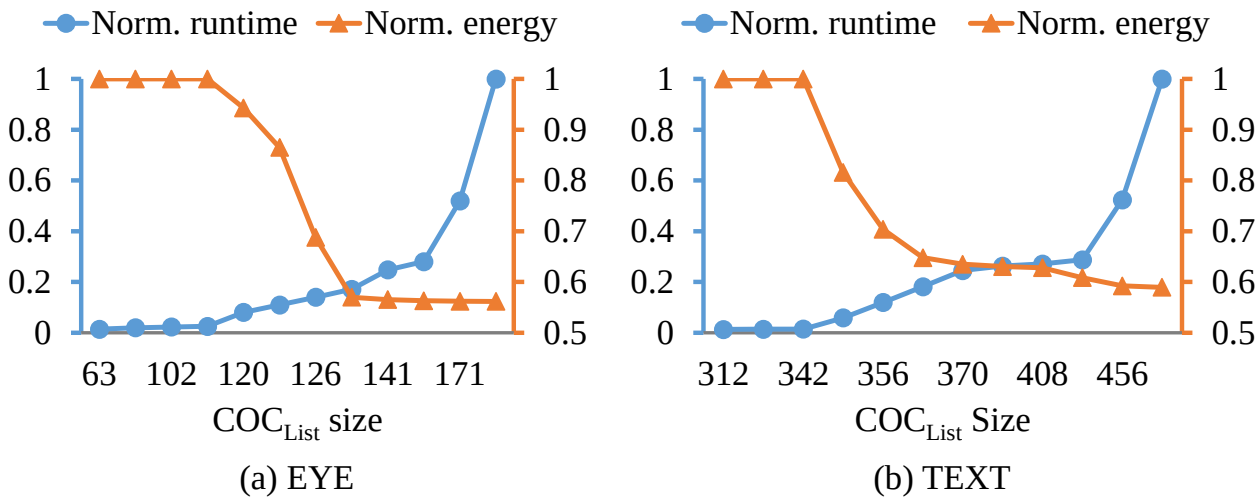


Figure 3.8. Runtime *vs.* optimality tradeoff analysis

3.7.4 Illustration of Clock Overgating in Action

Fig. 3.9 visually illustrates clock overgating in action by plotting the activity of FFs during around 1000 cycles of execution of the EYE application. In Fig. 3.9, each point on the Y axis corresponds to a distinct FF, while the X axis represents execution cycles. The color at each (x, y) coordinate indicates the status of FF y at cycle x. In a given cycle, an FF can be either active, clock gated or clock overgated, marked by blue, green and red colors respectively. Also, the length of the segment in the Y axis corresponding to each FF is proportional to the energy benefits possible by gating the FF. Therefore, the area in Fig. 3.9 occupied by the overgated regions roughly corresponds to the overall energy benefits. It is observed that only 12% of the FFs (FF index 162–183) are overgated in the circuit, but since they and the combinational logic that they feed contribute up to 50% of the total energy, ∼40% improvement in energy is achieved.

3.8 Summary

The intrinsic resilience in emerging applications provides new opportunities for optimizing hardware through approximate computing. This chpater proposes clock overgating, an RTL approximation technique. The key idea behind clock overgating is to suppress the clock signal to selected FFs in the circuit even when the circuit functionality is affected as

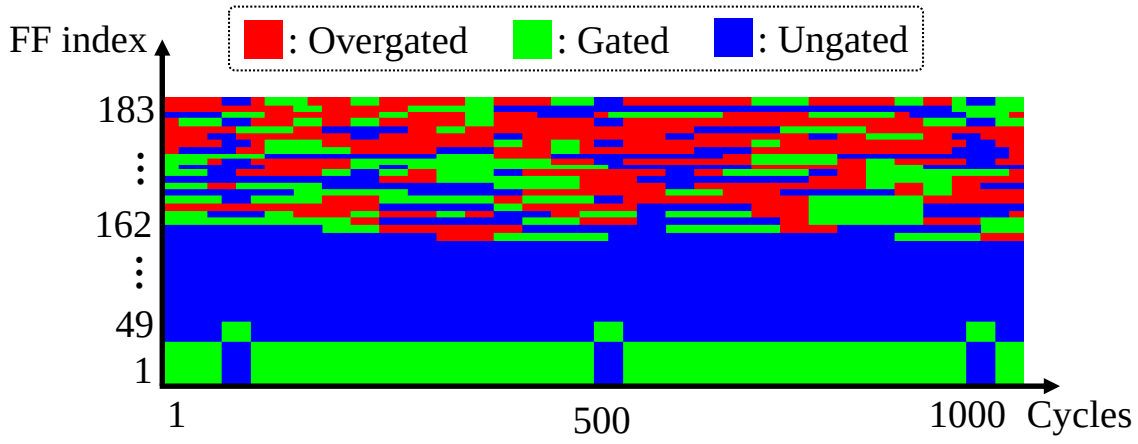


Figure 3.9. Clock overgating status over time

a result, thereby creating a trade-off between energy and quality. A systematic methodology is developed which identifies an energy-efficient clock overgating configuration given any arbitrary circuit and output quality constraint. The methodology employs three key strategies that leverage the semantic information available at the RTL to efficiently explore the large space of overgating configurations. The results across hardware implementations for a range of machine learning algorithms suggest that clock overgating is a promising approach to approximate hardware design and that it enables favorable tradeoffs between energy and output quality.

4. VALUE SIMILARITY EXTENSIONS FOR APPROXIMATE COMPUTING IN GENERAL-PURPOSE PROCESSORS

4.1 Introduction

In the previous decade, Approximate Computing (AxC) has emerged as a popular design paradigm that leverages the intrinsic ability of applications to tolerate approximations in some of their computations to boost compute efficiency [24], [84]. A significant fraction of prior efforts in AxC have focused on specialized architectures such as domain-specific processors [16], GPUs [17], or custom accelerators [84]–[86], while relatively limited effort has been devoted towards General-Purpose Processors (GPPs). However, modern applications that benefit from AxC are still widely executed on GPPs for various reasons, ranging from their easy programmability to stringent cost/area budgets precluding the use of custom accelerators. For example, the majority of Facebook's inference workloads run on CPUs within edge devices [87]. Hence, leveraging AxC to improve the efficiency of GPPs is of significant interest.

AxC in GPPs. A key challenge in approximating GPPs is that their execution units contribute only a small fraction of total energy. Hence, AxC in GPPs has been predominately achieved through software-level approximations such as function approximation, loop skipping, and relaxed synchronization [24], [71], [84]. The few efforts that propose hardware approximations for GPPs employ approximations either within the execution units through memoization [88], [89] and voltage scaling [39], [90], or in the memory sub-system through load value prediction [91]. The benefits from these approaches are limited by the energy expended in the parts that are *not approximated*, such as instruction fetch, decode, control, *etc.*

AxC through Value Similarity. This chapter proposes a holistic approach to AxC in GPPs that encompasses compute, memory and control front-ends. This chapter leverages the application property of *value similarity*, *i.e.,* input operands to computations that occur close-in-time take similar values, thereby producing results that are similar. An analysis on six representative Machine Learning (ML) workloads indicates that value similarity is

broadly prevalent, impacting as much as 80% of the computations executed. This provides an opportunity to pre-detect similar computations and skip fetch-decode-execute of entire instruction sequences, while substituting their results with those of previously executed computations, benefiting both performance and energy. To this end, this chapter proposes VSX, a set of lightweight micro-architectural and ISA extensions to leverage value similarity in GPPs. This chapter also presents compiler techniques that leverage user annotations to benefit from VSX in the context of common ML kernels.

A key trade-off in the design of VSX is balancing the window-of-opportunity *i.e.,* the span of computations across which similarity is exploited *vs.* the complexity of the implementation and overheads incurred. To ease the pressure on the memory-subsystem, GPP applications are structured to maximize access locality to elements within a cache-line. This chapter leverages this observation in VSX and pre-detect similar values among elements of a cache-line when performing a load (the similarity threshold is defined by software). The pre-detected similarity information is passed to the instruction fetch unit, which skips instructions even before they enter the pipeline by appropriately controlling the program counter.

VSX provides maximum flexibility to the compiler to define which instructions are skippable under what conditions. This is critical as the impact of approximations on the overall quality cannot be solely ascertained in hardware. To this end, the micro-architecture contains a programmable *Similarity Based Skip Table* that is configured before entering a code region where value similarity can be exploited. Based on programmer annotations in the code, the compiler generates the skip information for common ML kernels such as GEMM, Conv2D *etc.* As a further optimization, VSX also supports *iteration fast-forwarding*, *i.e.*, when operands to a set of loop iterations are similar to a previous one, the entire iteration set is skipped and replaced by a shorter specialized instruction sequence to produce an approximate output.

In summary, the key contributions of this chapter are:

- This chapter proposes a holistic approach to AxC in GPPs, targeting benefits in compute, memory and control front-ends, by leveraging value similarity in input operands across computations that occur close-in-time.

- This chapter proposes a set of lightweight micro-architectural and ISA extensions called VSX that enable dynamic pre-detection of similar computations to conditionally skip the fetch-decode-execute of entire instruction sequences. VSX also includes the ability for iteration fast-forwarding, wherein a set of loop iterations are replaced by a small, specialized sequence of instructions.

- This chapter develops compiler techniques that are guided by user-annotations and transform common application kernels to exploit value similarity and benefit from VSX.

VSX is implemented within a RISC-V in-order processor in RTL and synthesized to commercial 45nm technology. With an area overhead of 2.13%, VSX achieves 1.19×-3.84× speedup with <0.5% accuracy loss on 6 ML benchmarks.

4.2 Comparison to Related Work

Prior efforts on approximate computing that exploit value similarity as the source of efficiency improvement can be grouped into the following categories.

Approximate memoization. One way of exploiting value similarity is to save computation results for commonly occurring inputs and reuse them upon seeing similar inputs. This idea has been explored in the SW domain [69], the GPP domain [88], [89], [92], and the custom accelerator domain [18], [19]. This chapter's approach differs from these works by being more holistic, *i.e.*, not restricted to the execution units, and by eliminating the need for large memoization buffers.

Approximate load value prediction. Another line of work targeting value similarity focuses on similar values being loaded by GPPs [91] and GPUs [93]. They use the history of loaded values to predict the current load value in case of a cache miss. This approach effectively hides cache miss penalties in memory-bound situations where multiple processor

cores fetch data. On the contrary, this chapter's approach entirely skips loads as well as computes and other control instructions for data elements that have already been brought up to the L1 cache, thereby being more broadly applicable, including in compute-bound applications.

GPU-specific approximations. Prior efforts that exploit value similarity in GPUs include sharing an adjacent thread's result [73], or dynamically detecting inputs with similarity and computing a representative result for them [17]. This chapter's approach differs significantly due to the focus on GPPs. As mentioned earlier, workloads that can benefit from AxC often run on CPUs, especially in resource-constrained platforms where using a GPU is not a feasible option.

Input sampling. Exploiting value similarity among neighboring data elements, prior efforts redirect access to neighbor elements [94], or sample inputs such as pixels to reduce the number of loads [68]. In contrast to these pure SW techniques that apply approximation statically by assuming that neighboring elements are similar, the proposed HW approach detects similar values on-the-fly and approximates them so that the impact on output quality is minimized.

4.3 Value Similarity: Sources, Opportunities and Challenges

Value similarity refers to the presence of numerically similar values in neighboring elements of a data-structure. The two main sources of value similarity are: i) redundancy in real-world inputs, *e.g.*, homogeneous regions within an image, and ii) nature of the computation itself, *e.g.*, sorting algorithms placing similar values nearby or saturating activation functions in DNNs. Across the 6 ML benchmarks considered in this chapter's experiments, 78% of scalar values accessed are similar to within 20% of another element inside a window of 8 nearby elements. The incidence of similar values remains high even at tighter thresholds, or across smaller windows.

A large number of compute kernels access successive elements of their data-structures (*e.g.*, activation and weight arrays in DNNs) and perform the same operation (*e.g.*, multiply-and-accumulate) on them across multiple loop iterations. The presence of value similarity in these data-structures can thus lead to load instructions loading similar values and feeding

them to compute instructions (*e.g.*, addition, multiplication, *etc.*), which in turn produce similar results across consecutive loop iterations. Accordingly, some of these instructions can be skipped and their results can be approximated with that of a previous instruction to achieve execution time savings. However, realizing this approach in a GPP involves overcoming the following key challenges:

- *Identifying potentially skippable instructions.* Dynamically identifying instructions that produce similar results and are thus *skippable*, involves detecting and storing the similarity information of executed instructions. This can lead to a large overhead if done indiscriminately.

- *Detecting similarity before instruction execution.* For maximizing execution time savings, the skipped instructions need to be identified before even being fetched into the pipeline, thereby avoiding the introduction of bubbles. This demands the knowledge of an instruction's similarity information ahead of its execution.

- *Saving & reusing instruction results.* When an instruction is skipped, subsequent instructions using its result should be able to receive an approximated value. Thus, a mechanism for saving previous results and reusing them in place of skipped instructions is needed.

This chapter overcomes the above challenges through lightweight micro-architectural and ISA extensions to a GPP core that are described in the next section.

4.4 VSX: Value Similarity Extensions for General-Purpose Processors

This chapter proposes Value Similarity eXtensions (VSX), a set of low-overhead and minimally intrusive micro-architectural and ISA extensions, for exploiting value similarity in GPPs. This section presents the key ideas of VSX and illustrates in detail how they can be integrated within an in-order processor pipeline.

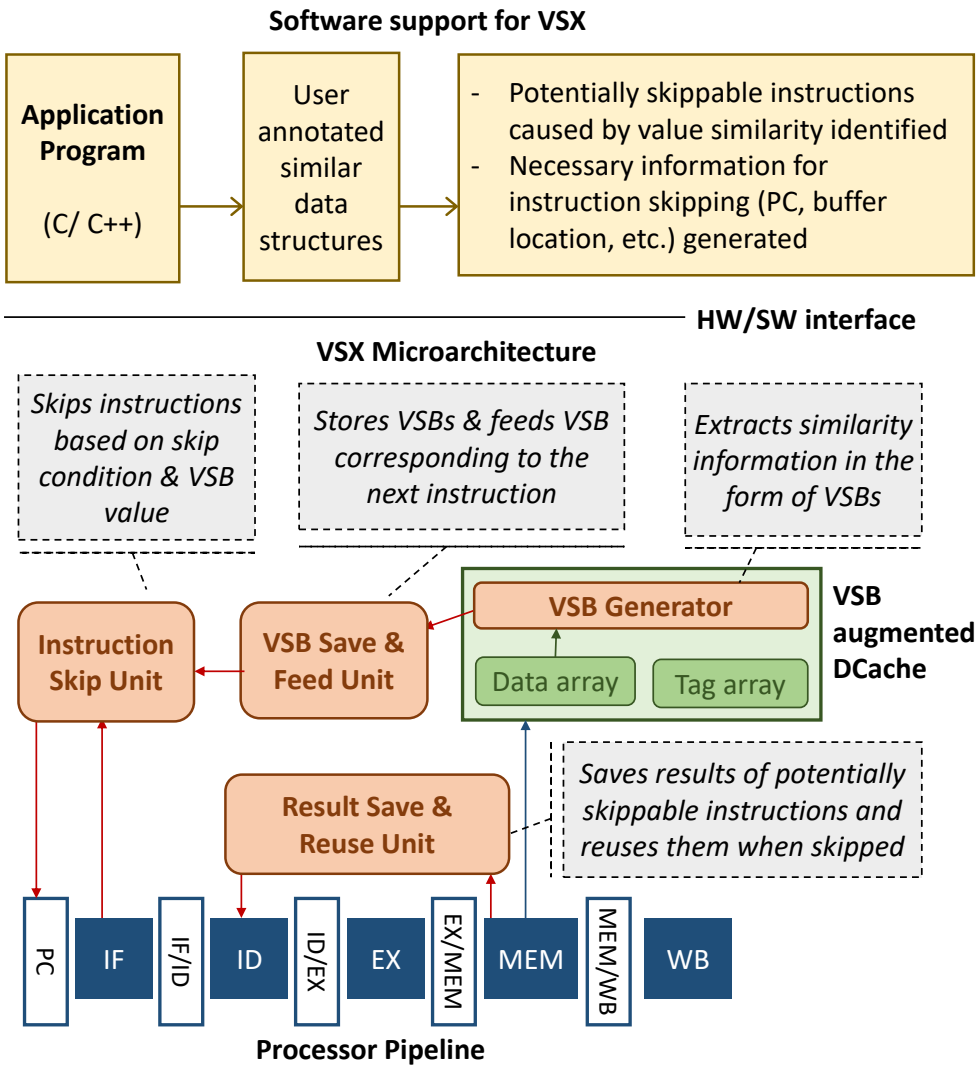


Figure 4.1. VSX overview

4.4.1 VSX: Overview

Figure 4.1 presents an overview of the proposed VSX extensions. These extensions help overcome the challenges listed in Section 4.3 through HW/SW co-design.

Identifying potentially skippable instructions from user annotations in software. This proposal provides a pragma that allows users to annotate data-structures with value similarity in a C/C++ program. From this annotation, a list of potentially skippable instructions are inferred at compile time to avoid the overhead of analyzing similarity of every instruction. This instruction list, along with their skip conditions, are stored in an *Instruction Skip Unit*

Code 4.1User annotation in SW

```
1 float *a, *b;
2 init(a, b, ARRAY_SIZE);
3 float dp = 0;
4 #pragma vsx(a,TH1,b,TH2) // by user
5 for (int i=0; i<ARRAY_SIZE; i++)
6     dp += a[i] * b[i];
```

(ISU) using a custom instruction. Before fetching every instruction, the *ISU* looks up this information to decide whether it can be skipped, in a similar way to [95].

Extracting similarity information from the data cache. This proposal augments the data cache with a *VSB Generator* that generates bit-flags called *Value Similarity Bits (VSBs)*, indicating whether each element in the cache-line is similar to the first element. When an application kernel traversing an array loads the first data element of a cache-line containing N elements, VSBs for the next $N-1$ elements are produced by the *VSB Generator*. These VSBs are transferred to the core and saved in a *VSB Save & Feed Unit*, which selects the appropriate VSB for a future load instruction and thereby identifies its similarity to a previous load. The similar loads are subsequently skipped by the *ISU*.

Saving & reusing instruction results in hardware. The proposed instruction skipping strategy requires instructions loading the first data element of each cache-line to complete execution, as VSBs are generated upon execution of those loads. The results of such loads are saved and then reused in place of a skipped load's result using a *Result Save & Reuse Unit (RSRU)*. Thus, compute instructions consuming a skipped load's result are guaranteed to receive a value similar to the actual one.

4.4.2 Program Annotations and Compiler Techniques for VSX

Code 4.1 shows how user annotation in SW can be done for a simple loop that computes a dot-product of two vectors. The programmer inserts a pragma right before the kernel of interest — the *for* loop in this example — that specifies the data-structures with value similarity (a, b) and the difference threshold for similarity evaluation (*TH1*, *TH2*). When the *for* loop in Code 4.1 is compiled to an assembly code shown in Figure 4.2, instructions *LD1*

and *LD2* are identified to be potentially skippable as they load data with value similarity. Compute instruction *MUL* also becomes skippable when *LD1* and *LD2* are both skipped. These skip conditions for different instructions are transferred to the *ISU* using a custom instruction, *SBST-LD*, before executing the application kernel.

4.4.3 Instruction Skipping & Result Reuse

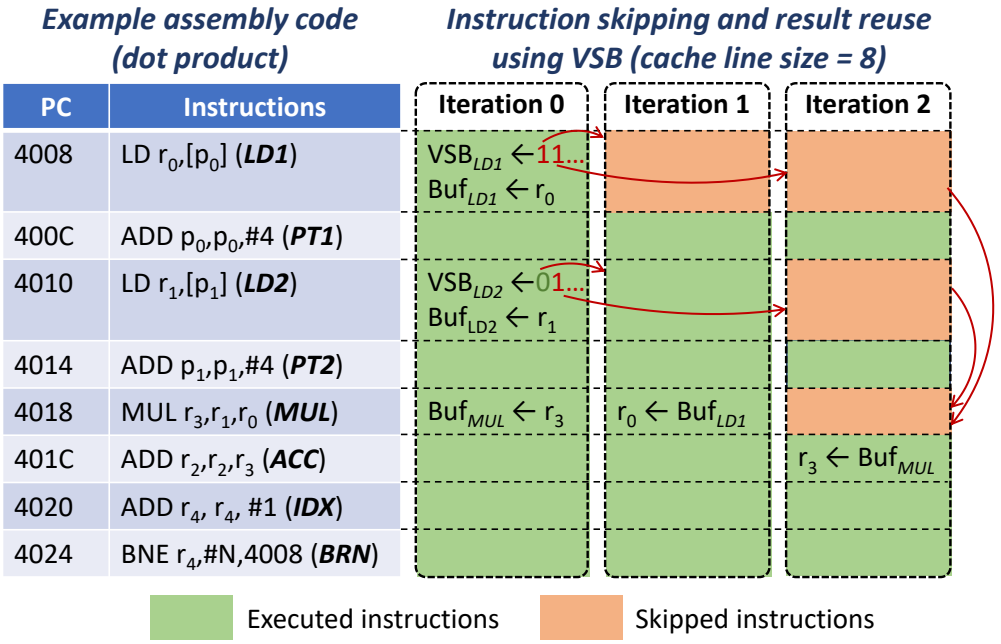


Figure 4.2. Instruction skip & result reuse example

Figure 4.2 illustrates instruction skipping & result reuse across multiple iterations of an example dot-product assembly code. As mentioned in Section 4.4.2, instructions *LD1*, *LD2*, and *MUL* are potentially skippable. In iteration 0, it is assumed that both *LD1* & *LD2* access the first element of a cache-line and the *VSB Generator* produces 7 VSBs each (VSB_{LD1}/ VSB_{LD2}) for the next 7 data elements (assuming a cache-line size of 8). Results of *LD1* & *LD2* are saved in their respective buffer slot (Buf_{LD1}/Buf_{LD2}) within the *RSRU* to be reused whenever *LD1* or *LD2* is skipped in the future. Result of *MUL* is also saved to Buf_{MUL} since it can be skipped as well. In iteration 1, *LD1* is skipped as its corresponding VSB_{LD1} equals 1. When *MUL* requires the result of *LD1* in the same iteration, the saved result Buf_{LD1} is used in

place of operand register r_0. In iteration 2, both *LD1* & *LD2* are skipped as both VSB_{LD1} and VSB_{LD2} equals 1. *MUL* is also skipped and *ACC* uses the value of Buf_{MUL} in place of its operand register r_3 as indicated.

4.4.4 VSX: Microarchitecture

Figure 4.3 shows the proposed micro-architectural extensions integrated within a 5-stage in-order processor pipeline. The details of each extension are presented below.

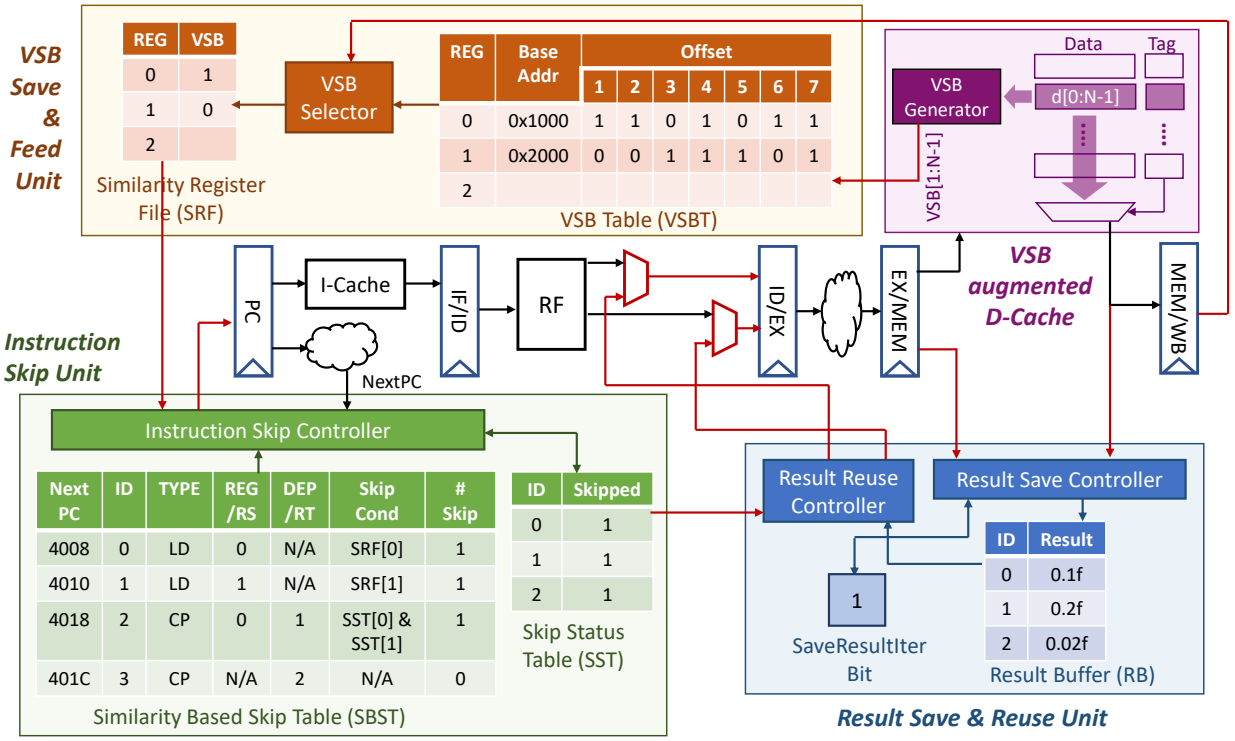


Figure 4.3. VSX microarchitecture

VSB Generator resides inside the data cache and generates VSBs for future load values. It performs a similarity check whenever a potentially skippable load instruction (identified by the *ISU*) accesses the first element of a cache-line. For a cache-line of N data elements (*d[0:N-1]*), $N-1$ VSBs (*VSB[1:N-1]*) indicating each data element's similarity to *d[0]* are generated. The value difference is compared with the user-defined threshold from Section 4.4.2 in order to evaluate similarity.

VSB Save & Feed Unit consists of a *VSB Table (VSBT)*, a *VSB Selector*, and a *Similarity Register File (SRF)*. The *VSBT* is a register file storing the VSBs generated by the *VSB Generator* for different cache-line addresses (*BaseAddr*). It is indexed using pointer registers (*REG*). Whenever a pointer register is updated, the *VSB Selector* uses the updated memory address to select the appropriate VSB from the *VSBT* and stores it in the *SRF*. The *ISU* refers to the *SRF* to obtain the similarity information for the next load instruction.

Instruction Skip Unit (ISU) consists of a *Similarity Based Skip Table (SBST)*, a *Skip Status Table (SST)*, and an *Instruction Skip Controller (ISC)*. The *SBST* stores the information necessary to identify skippable instructions, generated from user annotations described in Section 4.4.2. The individual entries correspond to different instruction regions, identified using the *NextPC* field, which become skippable whenever the conditions within the *SkipCond* field are met. For every instruction in the *IF* stage of the pipeline, the *SBST* is looked up with the *NextPC* value. If there's a match, the corresponding *SkipCond* is evaluated by the *ISC* and the PC is advanced by the value of the *#Skip* field. For a load instruction (*SBST.TYPE* equals *LD*), *SBST.REG* — index of the pointer register containing its target address — is used to access its corresponding VSB in the *SRF* while evaluating *SkipCond*. The *ISC* further records the result of this skip condition in the *SST* to allow skip condition evaluations of future instructions. For a compute instruction (*SBST.TYPE* equals *CP*), the *SBST.ID* of instructions producing its operands are used to access the *SST* for evaluating the skip condition.

Result Save & Reuse Unit consists of a *Result Buffer (RB)*, a *Result Save Controller (RSC)*, a *Result Reuse Controller (RRC)*, and a *SaveResultIter Bit*. The *RB* saves the results of potentially skipped instructions in different entries indexed using their *SBST.ID*s. The *RSC* in the *MEM* pipeline stage controls the write-enable condition of the *RB*. The result of a load instruction is allowed to be written to the *RB* if it corresponds to the first element of a cache-line. On the other hand, the result of a compute instruction is allowed to be written to the *RB* only if the *SaveResultIter Bit* is set. The *SaveResultIter Bit* is asserted whenever a load saves its result to the *RB*, indicating that a new value will be produced, and is cleared at the end of the loop iteration by the branch instruction (*i.e.*, *BRN* in Figure 4.2). It is ensured that all data-structures associated with potentially skippable loads are aligned

to the base address of a cache-line through standard C/C++ library macros. Finally, the *RRC*, in the *ID* stage of the pipeline, allows an instruction's operands to be replaced by the saved values in the *RB*. For any instruction, if the skip record of its operands (*SBST.RS* or *SBST.RT*) is set in the *SST*, the value of the corresponding operand is accessed from the *RB* instead of the register file.

4.4.5 Iteration Fast-Forwarding

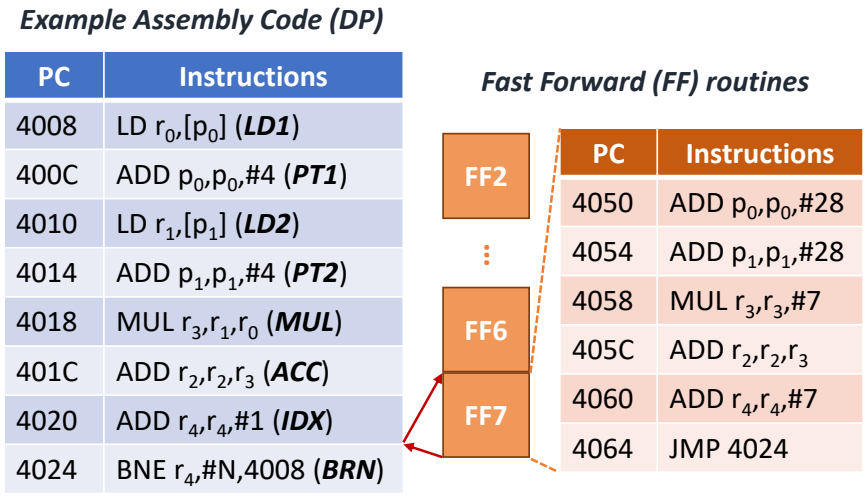


Figure 4.4. Iteration skipping example

In addition to the VSX microarchitecture shown in Figure 4.3 for skipping and reusing results of individual instructions, this chapter develops another optimization strategy based on two key observations. First, VSBs provide similarity information for multiple future iterations. For example, assuming that both loads (*LD1, LD2*) in the dot-product loop from Figure 4.4 have generated VSBs of 1111111, it is known that the next 7 loop iterations will produce similar *MUL* results. Second, multiple loop iterations producing similar results can be substituted with a specialized instruction sequence. For example, the next 7 iterations in the previous example can be substituted with a separate instruction sequence *FF7* that produces an identical output. This chapter proposes a systematic approach — iteration fast-forwarding — for detecting and substituting loop iterations that produce similar results. First, the run-length of VSBs for all loads is calculated upon each VSB generation. The

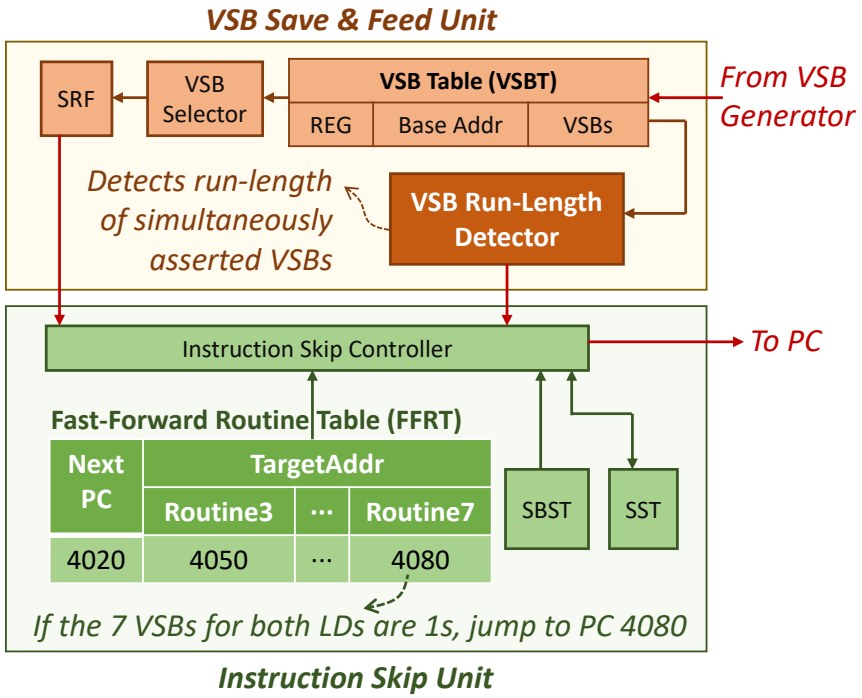


Figure 4.5. Modified *Instruction Skip Unit* and *VSB Save & Feed Unit* for iteration fast-forwarding

run-length information is then used to jump to the corresponding specialized sequence called *Fast-Forward Routine (FFR)* that substitutes multiple loop iterations and jumps back to the original instruction stream. Figure 4.5 shows the additional HW units for realizing iteration fast-forwarding. The *VSB Run-Length Detector* attached to the *VSB Save & Feed Unit* from Figure 4.3 detects the run-length of asserted VSBs for all entries in the *VSBT*. The *ISU* is now equipped with an *FFR Table*: A table storing the starting PCs of FFRs for different VSB run-lengths. Finally, the *ISC* modifies the PC to jump to an FFR corresponding to the VSB run-length.

4.5 Experimental Setup

Evaluation. This chapter implements VSX for a low-power single-core in-order RISC-V processor running at 200MHz. Details of the architecture are listed in Table 4.1. VSX is implemented in RTL (Register Transfer Level) using SystemVerilog HDL and synthesized to a commercial 45nm technology using Synopsys Design Compiler. The cache power and area

Table 4.1. Benchmark applications

Algorithm	Application	Dataset	# Inputs
GLVQ	Eye detection (EYE)	Image set from NEC labs	1465
KNN	Digit classification (DGT)	MNIST	1000
	Digit classification (DGT2)	Gissete	1000
SVM	Text classification (TXT)	Reuters	598
DNN	AlexNet (conv2, fc6)	ImageNet	1000
	VGG16 (conv1_2, fc6)		1000

were obtained using CACTI [96]. Compared to the baseline core and caches, **VSX exhibits 2.13% area and 1.15% power overhead**. The minimal overhead of VSX clearly enables its adoption in resource-constrained systems.

The proposed VSX microarchitecture is modeled using the gem5 [97] cycle-accurate architectural simulator. All benchmarks except DNNs were implemented in C++ and run directly on the gem5 simulator to obtain performance measurements as well as output accuracy. In case of DNNs, Caffe [98] — a widely-used deep learning framework — was coupled with gem5 in a manner that allowed performance measurement using gem5 and accuracy measurement using Caffe. Inputs & weights of the target layer were offloaded from Caffe to gem5 where matrix computations were performed, and their outputs were fed back to Caffe as the input of the next layer. All experiments were run in system-emulation mode.

Table 4.2. Gem5 system configuration

CPU	Single-core in-order RISC-V w/ FPU 200MHz clock speed
CACHE	L1D: 64KB 2-way SA (w/ prefetcher) L1I: 16KB 2-way SA 1-cycle hit latency, 32B lines
MEM	1GB LPDDR3

Benchmarks. To evaluate VSX, this chapter uses a benchmark suite consisting of 6 ML applications utilizing 4 different classification algorithms, listed in Table 4.2. For DNN benchmarks, VSX is applied to only the largest convolution and fully-connected layers *viz.*

$conv2/fc6$ for AlexNet, and $conv1\_2/fc6$ for VGG16, respectively. Classification accuracy *i.e.,* fraction of inputs classified correctly, is used as the quality metric.

4.6 Results

This section presents the results of this chapter's experiments to evaluate the effectiveness of VSX.

4.6.1 Speedup-Accuracy Tradeoff

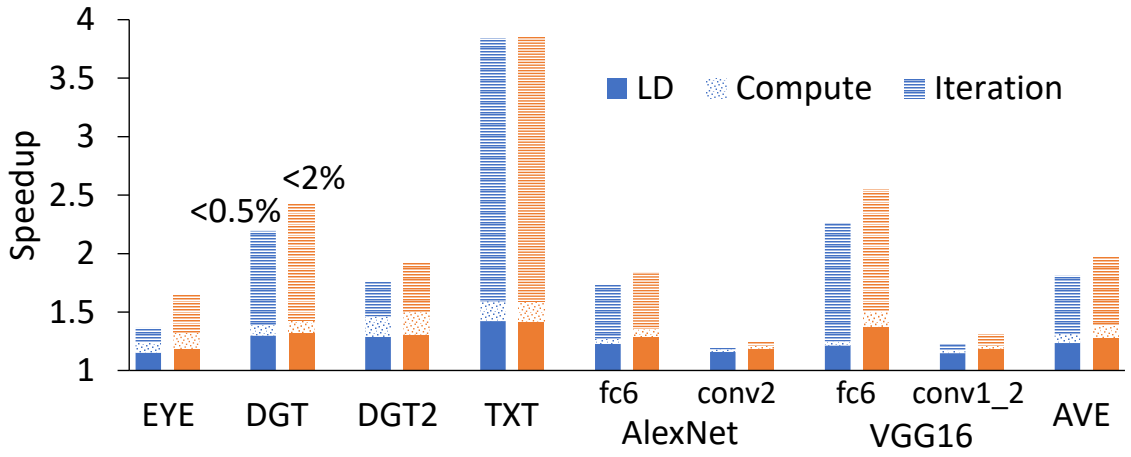


Figure 4.6. Speedup-accuracy tradeoff

Figure 4.6 shows the speedup achieved by VSX for different output quality constraints. The speedup is broken down into different components based on the type of instruction skipped: load, compute, or full iteration. For a negligible loss (<0.5%) in output quality, skipping loads results in a speedup of 1.15×-1.42× (average: 1.24×). Skipping compute instructions corresponding to the skipped loads increases the speedup to 1.17×-1.59× (average: 1.31×).

Finally, skipping iterations based on the run-length of asserted VSBs further increases the speedup to 1.19×-3.84× (average: 1.81×). For a relaxed quality constraint of <2%, the achieved speedup increases to 1.19×-1.42× (average: 1.28×) by skipping loads, 1.21×-1.59×

(average: 1.39×) by skipping loads and computes, 1.25×-3.84× (average: 1.97×) by skipping iterations as well.

4.6.2 Skip Rate Analysis

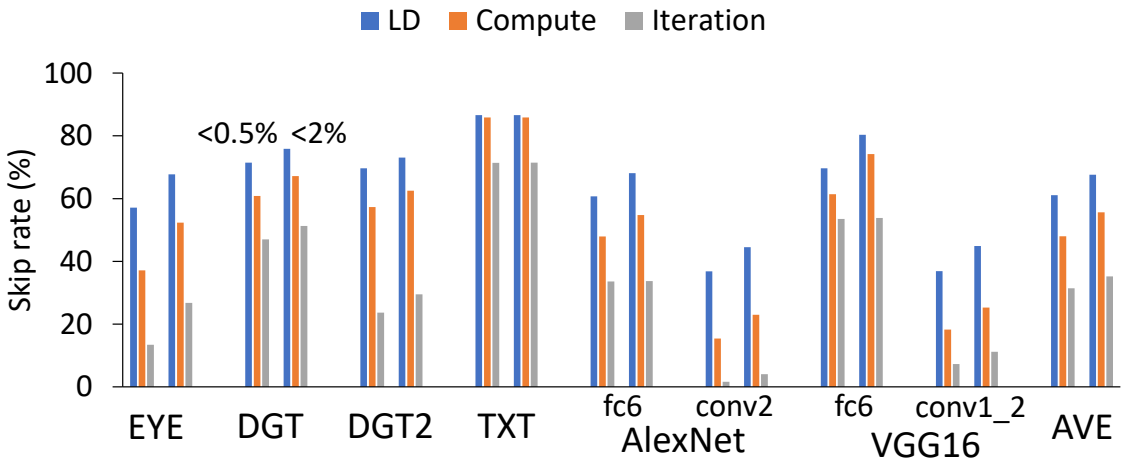


Figure 4.7. Skip rates for instructions & iterations

Figure 4.7 shows the skip rates for load/compute instructions and iterations under different output quality constraints. Under a tight quality constraint (<0.5%), skip rates range between 36.8%-86.6% (61.1% on average) for loads, 15.4%-85.9% (48.0% on average) for computes, and 1.6%-71.4% (31.4% on average) for iterations. When the quality constraint is relaxed to <2%, skip rates increase to 44.5%-86.6% (67.6% on average) for loads, 22.9%-85.9% (55.6% on average) for computes, and 4.1%-71.5% (35.2% on average) for iterations. *TXT* shows the highest skip rate possible for real-world inputs due to its extreme sparsity (>99%), while convolution layers of DNNs show low skip rate due to lack of similarity in their weights.

4.6.3 Speedup Across Different Spatial Distribution of Similarity

Figure 4.8 illustrates the effect of different spatial distributions of similarity on instruction & iteration skipping. This subsection varies the amount of similarity in the input data-

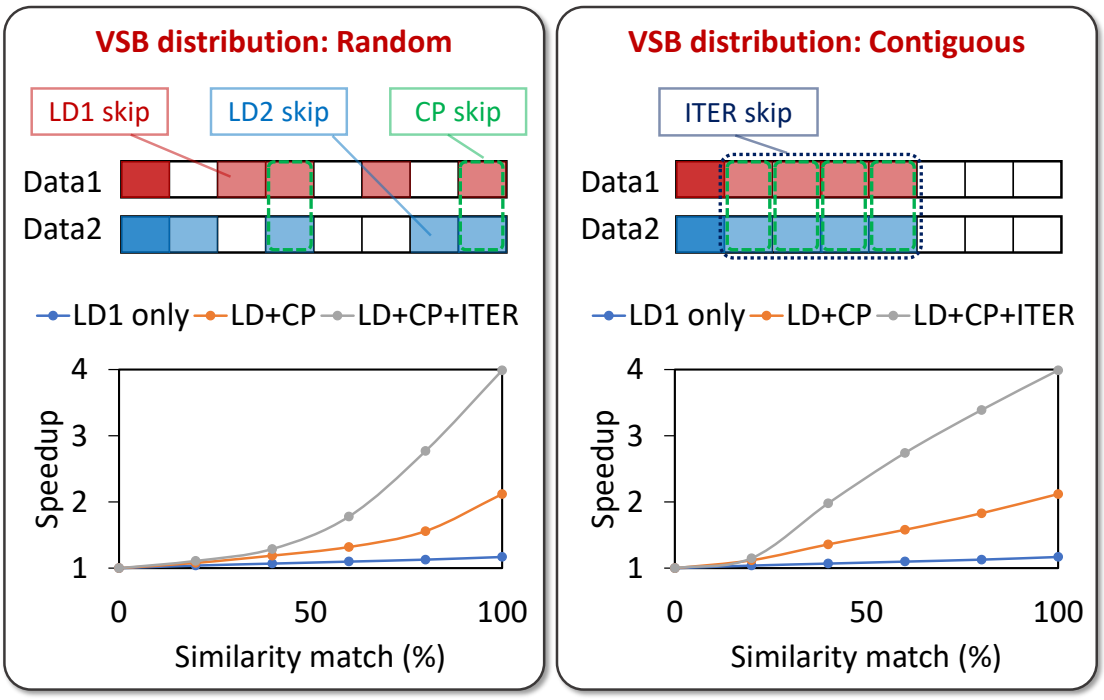


Figure 4.8. Speedup *vs.* similarity analysis

structures of a dot-product kernel and considers two different spatial distributions, *viz.*, *random* and *contiguous*.

Contiguous distribution maximizes compute and iteration skipping through the highest possible run-length of VSBs across all loads. The speedups observed for both distributions are similar at low similarity level (low probability of finding VSBs for all loads asserted) and at extremely high similarity level (almost all VSBs are asserted). However, at intermediate similarity levels, the speedup gap between them can be significant (*e.g.*, 1.78× for random vs. 2.74× for contiguous at 60% similarity).

4.6.4 VSX *vs.* Load Value Approximation

This subsection compares the speedup benefits of VSX with that of Load Value Approximation [91] (LVA) for <2% quality loss in Figure 4.9. LVA is an AxC technique that reduces the cache miss latency in GPPs by predicting a load value from previous load values (this subsection uses the average of 4 previous values). Every prediction is evaluated afterwards

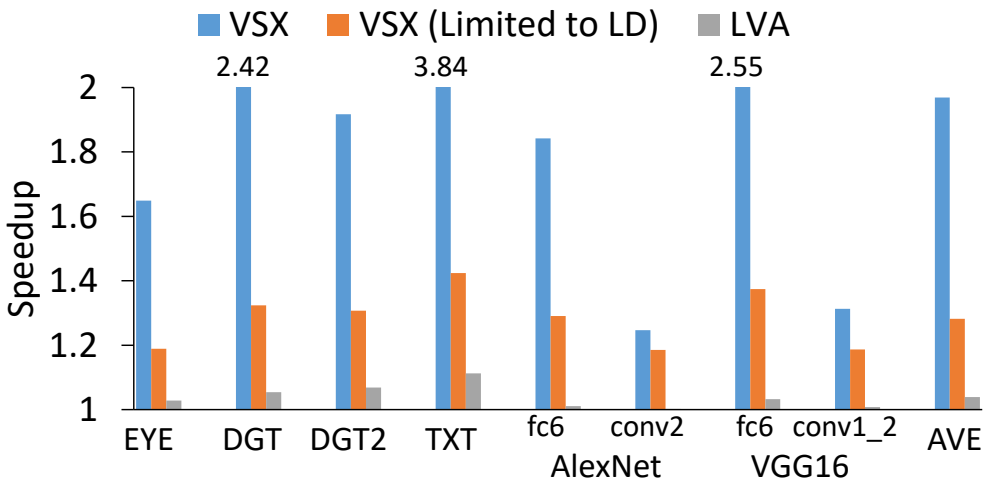


Figure 4.9. Iso-accuracy speedup comparison for VSX *vs.* LVA

by checking if the predicted value falls within a user-defined *confidence window* of the actual value. This confidence window in turn serves as a knob for modulating performance-accuracy trade-off. Since LVA targets only load instructions, this subsection also shows the speedups achieved by VSX when skipping is limited to only load instructions in Figure 4.9. It is observed that the average speedup for VSX in default mode (1.97×) and VSX limited to loads (1.28×) is significantly higher than LVA (1.04×).

For a slow single-core processor used in this chapter's experimental setup — along with the application kernels showing high spatial locality — cache miss rate is already too low (4.02%) for LVA to optimize. On the contrary, VSX is able to skip a wide range of instructions once the current cache-line is brought up to the L1 data cache, which effectively translates to speedups.

4.7 Summary

Approximate computing in General-Purpose Processors (GPPs) requires a holistic approach targeting benefits in compute, memory and the control front-ends. This chapter proposes VSX (Value Similarity eXtensions) — a set of lightweight micro-architectural and ISA extensions for GPPs — that exploits value similarity within data-structures for performance improvement. The key idea is to pre-detect similar values in the granularity of a cache-line

and use that information to skip fetch-decode-execute of instruction sequence and/or loops, and approximate their result with a previously saved one. This chapter presents compiler techniques to transform common Machine-Learning (ML) kernels to use VSX. This chapter evaluates VSX on a low-end in-order RISC-V processor platform and shows 1.19×-3.84× speedup across a set of ML benchmarks at the cost of a 2.13% area overhead, highlighting its applicability to edge platforms.

5. DATA SUBSETTING: A DATA-CENTRIC APPROACH TO APPROXIMATE COMPUTING

5.1 Introduction

Emerging workloads related to machine learning and data analytics occupy a significant fraction of execution cycles across the entire spectrum of computing platforms from mobile devices to the cloud. These workloads process large amounts of data and thereby place immense demands on the memory sub-systems of modern computing platforms. With the growing processor-memory performance gap (exacerbated by many-core processors and hardware accelerators with thousands of cores), the ability of the memory sub-system to feed data to the processing cores has become the key determinant of overall system performance.

Approximate Computing (AxC) is an emerging design paradigm that leverages intrinsic resilience to improve efficiency [1], [24]. Over the years, AxC techniques spanning various levels of design abstraction — software, architecture and circuits — have demonstrated significant improvements in performance and energy across a broad range of applications. A vast majority of prior efforts in AxC have focused only on approximating *computations* [5], [9], [15], [16], [20], [21], [23], [71], [73], [75], [90], [99]. However, recent work suggests that applying it to other system components has a potential to result in additional improvements [91], [93], [100]–[103]. This work explores AxC as an approach to alleviating the memory bandwidth bottleneck.

This chapter proposes a *data-centric approach* to approximate computing that can be used to improve the performance of software on off-the-shelf computing platforms (without any hardware changes). The key idea is to *modulate accesses to data structures so as to shape the memory traffic* such that the overall memory bandwidth requirement is reduced. Specifically, a technique called *data subsetting* is proposed, wherein accesses to the data structure are restricted to a subset of its elements. Constraining the data structure accesses to lie within a smaller foot-print renders the resulting memory traffic more cache-friendly, thereby enhancing performance.

This chapter identifies two key challenges with data subsetting. First, a subset of elements that are representative of the entire data structure needs to be identified. The size of the

subset and the elements constituting the subset need to be selected based on application and data characteristics. These choices could vary from program-to-program, across data structures within the same program, and even for the same data structure over the course of execution of a program. The second challenge pertains to how the data accesses are approximated. When the application program attempts to access data that falls outside the chosen subset, the access needs to be redirected to a location within the subset. To address both these challenges in a manner that is transparent to hardware while requiring only minimal changes to the application program, this chapter defines a new templated data structure called `SubsettableTensor`. A `SubsettableTensor` is associated with an Access Redirection Function (ARF) through which the application program specifies the portion of the data structure that can be accessed and how accesses to other parts of the data structure are handled.

To extract maximum performance benefits from data subsetting on off-the-shelf computing platforms, the following optimizations are incorporated. First, to exploit spatial locality, caches are accessed at the granularity of cache lines where each cache line could store multiple elements from the data structure. If all elements present within the cache line are not part of the accessible subset, then the benefits of data subsetting are not fully realized, as some bandwidth is wasted in fetching elements that will never be accessed. To avoid this, *Subset Buffer* is proposed into which all the elements in the accessible subset are copied. During execution, accesses to the data structure are translated to a location within the subset buffer. Also, since data subsetting redirects many accesses to the same location in memory, some of the computations could be rendered redundant. This chapter provides a mechanism for the application software to identify and eliminate these redundant computations to further enhance performance benefits.

In summary, the key contributions of this work are:

- Proposes a data-centric approach to approximate computing, which shifts the focus of approximations from computations to data by modulating accesses to data structures so as to reduce memory bandwidth. Presents an approximation technique called

data subsetting, wherein accesses are limited to a subset of the data structure, which enhances performance by making the memory traffic more cache-friendly.

- Realizes data subsetting in a manner that is minimally intrusive to application software through extensions to data structures. Specifically, develops a templated data structure called `SubsettableTensor`. This approach enables the application program to fully control which elements of the data structure are part of the accessible subset and how accesses to other elements are handled.

- Improves the performance of data subsetting on off-the-shelf platforms through the use of a *subset buffer*, which enhances spatial locality among elements in the accessible subset. Also enables application software to eliminate computations that are rendered redundant as a result of data subsetting.

Data subsetting is applied to parallel software implementations of 7 machine learning applications. On a 48-core AMD Opteron server, a performance improvement of 1.33×-4.44× was achieved with negligible loss in output quality.

5.2 Comparison to Related Work

Section 2.3 presented approximate SW design techniques which are mostly compute-centric – which means that their focus is on performing computations in an efficient manner. Recent work has begun to recognize the merits of applying approximations to other system components [103]. Specifically, a few recent efforts have applied AxC to the memory sub-system. These include reducing DRAM refresh rate [100], [104], storing/accessing data in a compressed format [102], and speculating on the results of loads [91], [93], among others. Data subsetting is qualitatively different from the above techniques, which approximate the value of the data accessed (through skipped refreshes, compression, *etc.*) as opposed to approximating the memory location that is being accessed. Further, all the above methods require changes in hardware to carry out approximations. On the other hand, data subsetting is realized using off-the-shelf hardware and with minimal changes to application software.

5.3 Data Subsetting

The goal of this work is to improve the performance of memory-bound applications on off-the-shelf computing platforms using approximate computing. The key idea is to use a data-centric approach, wherein the memory traffic is shaped by modulating data accesses in a manner that renders them more cache friendly. This is achieved by proposing a new approximation technique called data subsetting. This section begins by describing the key concepts behind data subsetting and outline the key challenges in its realization. Finally, optimizations that enhance the benefits of data subsetting on off-the-self computing platforms are described.

5.3.1 Data Subsetting: Concept

Figure 5.1 illustrates the concept behind data subsetting. The key idea is to *approximate the accesses* to the elements of a given data structure. Consider a data structure D, which is a set of elements stored in memory. Let D_{subset} be a subset of elements in the data structure. When the application attempts to access (read or write) an element of D, data subsetting approximates the access so as to ensure that it falls within D_{subset}. If the element to be accessed is already within D_{subset}, then no approximation occurs. If the element lies outside D_{subset}, then the access is *redirected* to a different element that falls within D_{subset}. In the case of a read request, an incorrect value is returned, whereas in the case of a write, an incorrect memory location is written.

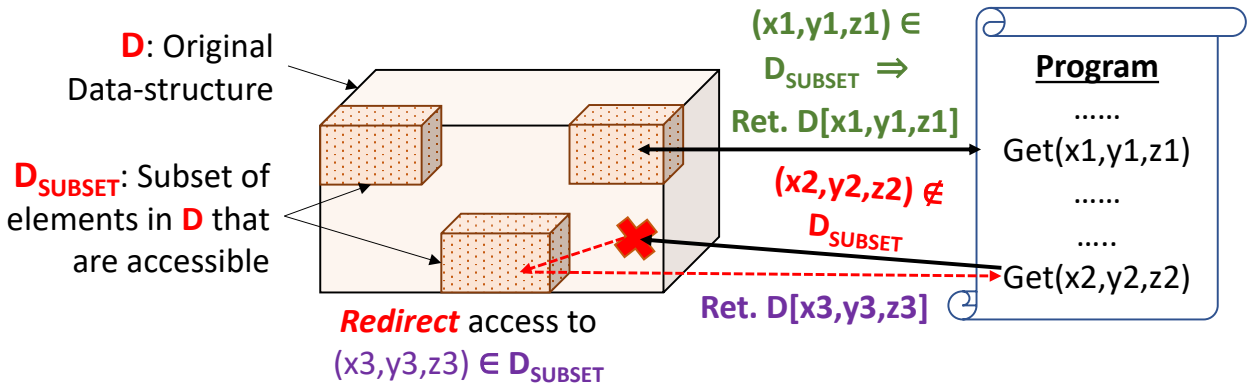


Figure 5.1. Data subsetting: Concept

Benefits. By limiting accesses to a subset of elements in the data structure, data subsetting results in the following benefits: (i) Since data subsetting limits the foot-print of data accesses, it reduces the size of the application's working set. This leads to fewer accesses to lower levels of the memory hierarchy, and in the extreme, could result in the entire working set fitting in on-chip cache, leading to substantial performance improvements. (ii) As accesses are redirected to fewer memory locations, it fundamentally increases locality and the FLOPs/Byte ratio of the application and reduces its total bandwidth requirement. (iii) Finally, since data subsetting returns the same value for many different reads and/or overwrites the same location for many different writes, some of the computations in the application are rendered redundant. Such computations can be identified and eliminated to further boost the overall performance (Section 5.3.3).

5.3.2 Data Subsetting: Challenges

This subsection describes the challenges in realizing data subsetting in practical applications.

Subset Selection

The first challenge lies in identifying the subset of data that is representative of the entire data structure. Naturally, subset selection is a strong function of the type of data processed by the application and the context in which they are used. For example, for data elements that are spatially correlated (*e.g.*, image pixels) or temporally correlated (*e.g.*, audio samples, video frames), a subset identified by *periodic selection* of elements often works well as it takes advantage of the locality in values. Different data strctures may exhibit spatial/temporal correlations at different granularities, which influences the subset selection. Figure 5.2(a) illustrates subset selection at the granularity of pixels (darker pixels are in the subset), exploiting the fact that computation results on neighboring pixels are similar. In contrast, Figure 5.2(b) shows a coarser-grained selection of a subset of images from a set of reference images (images with darker lines are in the subset), exploiting similarity across images. In

the case of data structures with unordered data elements, as shown in Figure 5.2(c), random subset selection may be desirable since it preserves the statistics of the data.

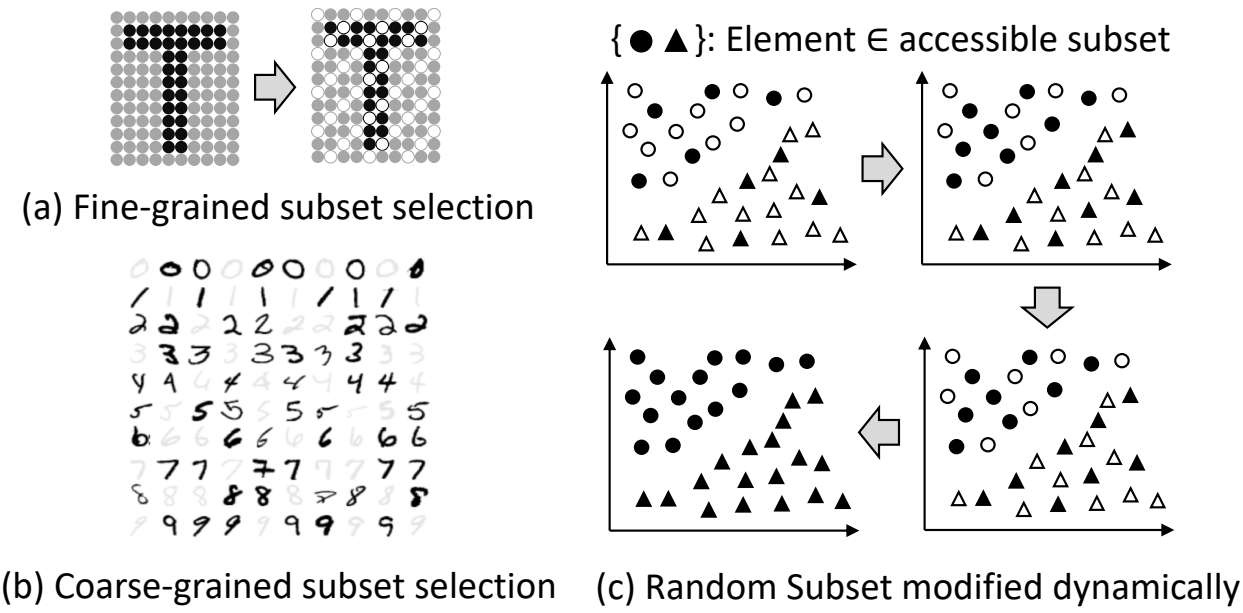


Figure 5.2. Different forms of subset selection

In some applications, the choice of the subset for a given data structure may vary dynamically through the course of program execution. For instance, iterative algorithms (K-Means clustering, Stochastic Gradient Descent for Deep Neural Network (DNN) training, *etc.*) compute the output by iterating over the same data multiple times, producing a more refined output in each iteration. In such cases, the initial iterations could be executed on a small subset of the data to compute an approximate value of the output. As the iterations progress, the size of the subset could be gradually increased, allowing the algorithm to refine the final value of the output.

Access Approximation through Redirection

Once the subset is identified, the second challenge lies in identifying how accesses to other parts of the data structure are approximated to fall within the subset. In this case, an *Access Redirection Function* (ARF) is defined, which is a many-to-one function that maps an index to an element in the data structure to an approximate index that falls within the subset.

Figure 5.3 illustrates ARF in the context of a 1D-tensor (D), where the subset (D_{subset}) is constructed by periodically sampling D at an interval K. Accesses are approximated such that all accesses between indices $i * K$ to $i * (K + 1) - 1$, where $i \in \{0, 1, \ldots, |D|/K\}$, are redirected to index $i * K$. In this case, the ARF is a staircase function with K being the height of each step. This can be mathematically expressed as $ARF(i) = K * \lfloor i/K \rfloor$.

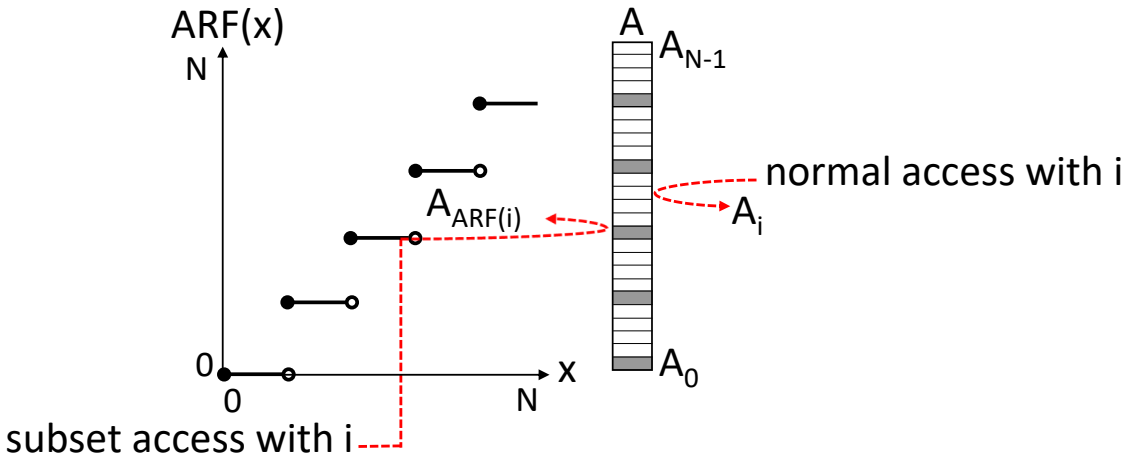


Figure 5.3. Access Redirection Functions

The objective of this chapter is to realize data subsetting with no changes to the hardware, while minimizing changes to the application software. This is achieved by building a templated data structure called `SubsettableTensor`, which is described in Section 5.4. A `SubsettableTensor` provides in-built implementations of commonly used ARFs, but also allows application software to specify any arbitrary ARF. The ARF may be modified over the course of program execution.

5.3.3 Data Subsetting: Optimizations

To maximize the performance benefits from data subsetting on off-the-shelf computing platforms, this chapter proposes 2 additional optimizations *viz. subset buffer* and *redundant computation elimination*, which are described in the following subsections.

Subset Buffer

In general-purpose systems, to exploit spatial locality in accesses, the cache hierarchy is accessed at the granularity of cache lines. It is possible for each cache line to store multiple elements of a data structure. This adversely impacts data subsetting when all elements present in a given cache line are not part of the accessible subset. This is because an access request to a subset element fetches the entire cache line (including out-of-subset elements) from lower in the memory hierarchy, amounting to wasted memory bandwidth. For example, as shown in Figure 5.4, consider a 1D-tensor with 8B data elements. Suppose the subset is constructed by periodically selecting one in every four elements. If each cache line is comprised of 8 data elements (64B), accessing a subset element would fetch all the non-subset elements surrounding it, which results in *no* bandwidth reduction.

To address this issue, once the subset elements are identified, a different space in memory called the *subset buffer* is allocated and it is filled with elements that are only part of the subset. Then, instead of accesses being redirected to subset elements within the data structure, they are redirected to a location within the subset buffer. For example, the ARF for periodic selection is modified as: $ARF_{sb}(\mathrm{i}) = ARF(\mathrm{i})/K = \lfloor \mathrm{i}/K \rfloor$. The cache lines that store the subset buffer are comprised only of subset elements, and no bandwidth is wasted. The performance overhead of populating the subset buffer for the first time is negligible, as the cost is amortized over several accesses to each location within the subset buffer.

Besides boosting performance, the use of subset buffers has other indirect benefits. Since subset elements are copied to a different location in memory, their values can be modified to better represent the data structure. For instance, consider a *neighborhood average* subsetting scheme, wherein the subset selection is periodic (period = K), but instead of picking every K^{th} element, the average value of the data elements within that period is used. In this case, *both* the access as well as the value are approximated together. Also, the use of subset buffers makes it easier to utilize optimized libraries (*e.g.*, BLAS) that expect their inputs to be laid out contiguously, by simply passing them a pointer to the subset buffer.

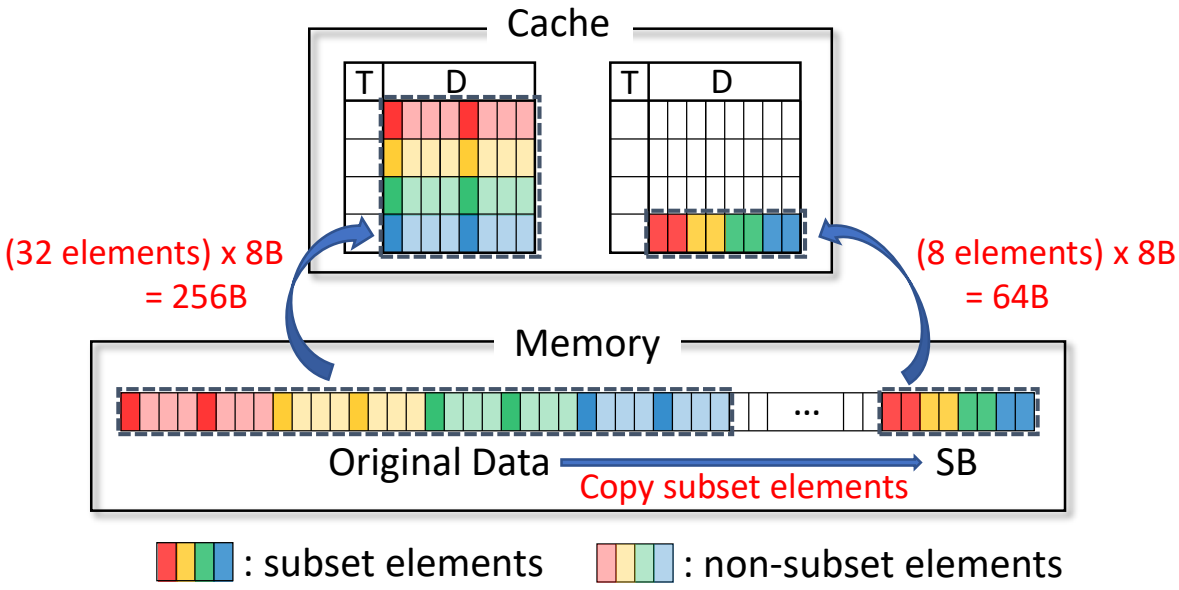


Figure 5.4. Memory bandwidth reduction by using subset buffer

Eliminating Redundant Computations

In data subsetting, since accesses to non-subset elements are redirected to subset elements, the computations that use the non-subset elements could be rendered redundant. For example, if the application computes the maximum value of all elements in the data structure, under data subsetting, iterating over the entire data structure (with access redirection) *vs.* iterating only over the subset elements would produce identical results. In many other cases, iterating just over the subset elements may produce an incorrect answer, but can be easily compensated to compute the correct result. For instance, computing the sum of all elements in the data structure under periodic data subsetting can be achieved by computing the sum over the subset and multiplying the result by the subsetting period.

One broad approach to enable applications to skip computations pertaining to non-subset elements is to provide a mechanism that iterates *only* through the data elements in the subset. To this end, this chapter develops a *custom iterator* for the `SubsettableTensor` data structure, which loops over the data elements presently in the subset. The application can use the iterator to compute only on the subset elements (with/without compensation at the end), further improving performance.

Code 5.1SubsettableTensor data structure

```
1  template <class T> class SubsettableTensor {
2     T *original_; vec<int> shape;
3     T *SB_;
4     // Type of ARF:  (i) Periodic, (ii) Random,
5     // (iii) Neighbor-average, (iv) User-defined
6     string ARFtype; vec<int> ARFparam;
7     int *userARF(int); // Func.ptr to custom ARF
8
9     SubsettableTensor(T* data, vec<int> shape);
10    Subset(ARFtype, ARFparam, int(&usrARF)(int));
11    int ARF(int index)
12     { /* Execute pre/user-defined ARF */ }
13    T& operator[] (const int index)
14     { return SB_[ARF(index)]; }
15 }
16 #define for_subset(ST, INDEX) \
17   /* iterate through subset indexes */
```

5.4 Realizing Data Subsetting in Software

Recognizing that applications in the domains of interest ubiquitously utilize multi-dimensional arrays or tensors to represent data, this chapter develops `SubsettableTensor`, a new templated data structure that application programs can use to realize data subsetting. This section describes the `SubsettableTensor` data structure and illustrates its use through a sample application.

5.4.1 SubsettableTensor Data Structure

Code 5.1 describes the `SubsettableTensor` data structure. It contains a pointer to the original data (*original_*) and a vector describing the shape of the tensor (line 2). It also contains a pointer to the subset buffer (*SB_*), which is populated when a subset is constructed. The data structure allows the application program to choose one of the pre-defined ARFs (*ARFtype* and *ARFparam*), or provide a custom ARF through a function pointer (userARF) (lines 4-7). This chapter's implementation supports 3 pre-defined ARFs *viz. periodic*, *random* and *neighbor-average* that were described in Section 5.3.1. The selection of

Code 5.2Data subsetting applied to K-Means clustering

```
1  int in[VEC][FEA], cent[K][FEA], assign[VEC];
2  SubsettableTensor<int> insub(in,{VEC,FEA});
3  insub.Subset(Random,{initSubsetRatio,1},NULL);
4  do {
5    int assign_old [VEC] = assign;
6    for_subset(insub, int i) {
7      int dist[K] = 0;
8      for(int j=0; j<K; j++) {
9        dist[j]=CalcDist(in[i,:], cent[j,:]);
10     int closest=FindClosestCluster(dist, j); }
11   fracReassigned = diff(assign_old, assign)/VEC;
12   newSubRatio = getSubRatio(fracReassigned)
13   insub.Subset(Random,{newSubRatio,1},NULL);
14   CalculateCentroid(in, cent, assign);
15 } while (fracReassigned > 0 && newSubRatio == 1)
```

ARF is achieved through a member function called *Subset* (line 10), which also allocates and fills in the subset buffer. Every access to a given index of the data structure is first passed through the ARF (lines 11-12), which computes an approximated index. The subset buffer is accessed using the approximated index and the value is returned (lines 13-14). Finally, to enable redundant computation skipping, `SubsettableTensor` provides an iterator (*for_subset*), which loops through the subset indices.

5.4.2 Illustration: K-means Clustering

This subsection illustrates how `SubsettableTensor` can be used in a practical application by using K-Means clustering as an example. In K-Means, given a set of points (represented as vectors), the objective is to group them into K clusters. K-Means is an iterative algorithm, where in each iteration, the distances between all points and the centroids of the K clusters are computed. Each point is assigned to the cluster to whose centroid it is closest. Each cluster's centroid is recomputed by averaging the points assigned to it. This process is repeated until convergence (*e.g.*, no point changes clusters).

Code 5.2 shows the pseudo-code for K-Means clustering with data subsetting (changes made for data subsetting are highlighted in boldface). In this case, the input points (*in*) are

defined as a `SubsettableTensor` – i$nsub$ (line 2). First, a subset is constructed by randomly selecting a fraction (i$nitSubsetRatio$) of the input points (line 3). Each iteration processes the chosen subset of points (line 6), computes the distances with each cluster centroid (lines 7-9), and assigns each point to the cluster whose centroid is closest to (line 10). Then the fraction of inputs whose clusters were reassigned ($fracReassigned$ in line 11) is computed, based on which a new subset ratio ($newSubRatio$ in line 12) is computed. When only a small fraction of inputs are reassigned, it is ascertained that the centroids have stabilized and therefore the size of the subset is increased to consider more points. This is achieved by calling the $Subset$ function on i$nsub$ with $newSubRatio$ (line 13). The algorithm terminates when there are no reassignments and the subset ratio is 1 *i.e.,* clusters have been assigned to all input points (line 15).

Thus, data subsetting can be realized using `SubsettableTensor` with minimal changes to the original program.

5.5 Experimental Setup

This section describes the experimental setup for evaluating data subsetting on a commodity platform.

5.5.1 Benchmarks

To evaluate data subsetting, this chapter considers a benchmark suite comprising of 7 machine learning applications listed in Table 5.1. The benchmarks use 5 different classification and clustering algorithms. In the case of the Deep Neural Network (DNN) benchmarks, data subsetting is applied only to the largest convolutional and fully connected layer *viz.* $conv2$ and $fc6$ for AlexNet, and $conv1\_2$ and $fc6$ for VGG16 respectively. For the classification benchmarks, classification accuracy *i.e.,* fraction of inputs classified correctly, is used as the quality metric. In the case of K-Means clustering, quality is measured as the average distance between all data points and their corresponding cluster centroids.

Table 5.1. Machine learning benchmark applications

Algorithm	Application	Dataset	# Inps	Data (MB)
GLVQ	Eye detection (EYE)	Image set from NEC labs	1465	64
KNN	Digit classification (DGT)	MNIST	1000	180
	Digit classification (DGT2)	Gissete	1000	115
SVM	Text classification (TXT)	Reuters	598	34
DNN	AlexNet	ImageNet	1000	3.9 (conv2)
				146 (fc6)
	VGG16	ImageNet	1000	110 (conv1_2)
				397 (fc6)
KMEANS	K-means clustering (KMS)	Volcanoes	128	512

5.5.2 Performance Evaluation

All the benchmarks were implemented in C++ and parallelized using OpenMP. This chapter also developed data-subsetted versions of the benchmarks by implementing the `SubsettableTensor` data structure in C++. This chapter's experiments were conducted on a 48-core server platform, whose parameters are shown in Table 5.2.

Table 5.2. System configuration used in experiments

CPU	AMD Opteron, 48 cores, 2.3GHz
CACHE	L1 64KB, L2 512KB, L3 20MB, 64B lines
BUS	42.7 GB/s peak bandwidth
MEM	190GB, DDR3-1333MHz, 8 channels

5.6 Results

This section evaluates the effectiveness of data subsetting using various experiments.

5.6.1 Performance Benefits

Figure 5.5 shows the speedup achieved with data subsetting for different application-level quality constraints. Across all benchmarks, the speedup ranges between 1.33×-4.44× (2.31×

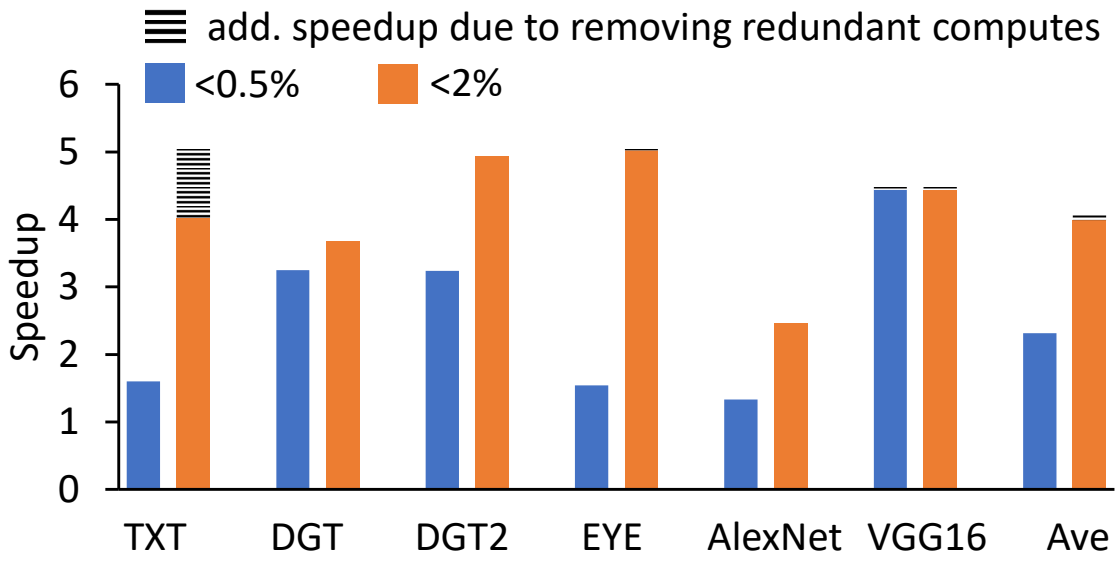


Figure 5.5. Speedup obtained within different quality constraints

on average) for a negligible loss (<0.5%) in quality. When the quality constraint is relaxed to <2%, the speedup increases to 2.47×-5.06× (4.05× on average). Figure 5.5 also highlights the performance gain due to skipping computations rendered redundant by data subsetting. Since the applications were primarily memory-bound, a significant fraction of the benefits stem from bandwidth reduction as opposed to computation skipping, which was effective only for `TXT` and on average yielded 7% additional performance gain.

5.6.2 Comparison with Loop Perforation

This subsection compares the performance of data subsetting with a well-known compute-centric approximation technique called *loop perforation* [71], wherein iterations of loops are periodically or randomly skipped from execution. Figure 5.6 shows the speedup achieved with increasing subsetting/perforation ratio (which varies inversely with the subset size or loop iterations executed). Computations on non-subset elements were eliminated using the for_subset iterator for data subsetting. First, in the case of data subsetting, as the subsetting ratio is increased, a steady improvement in performance is achieved. In some cases, the performance improvement is super-linear (*e.g.* subsetting ratio of 3 yields 3.5× performance benefits), as data structures begin to fit within levels of the memory hierarchy.

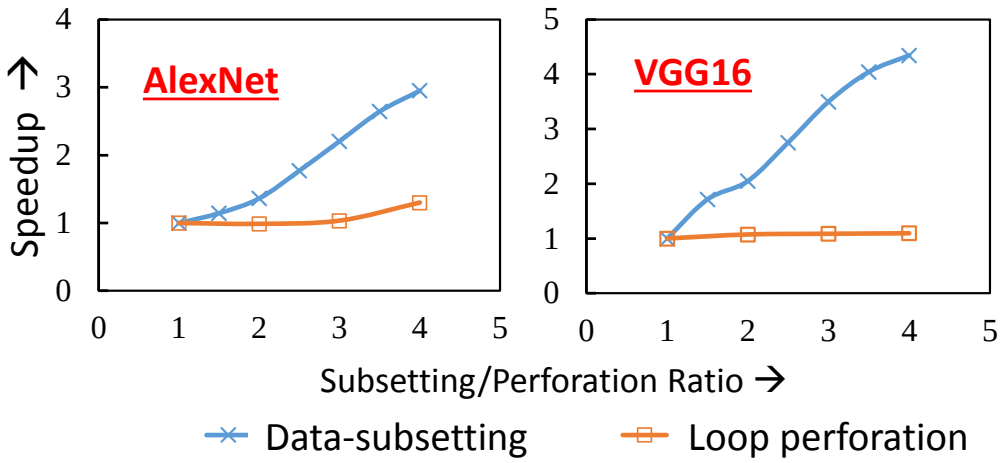


Figure 5.6. Data subsetting *vs.* loop perforation

In the case of loop perforation, the speed-ups are significantly lower. Although loop perforation eliminates both computations and data accesses present in the skipped loop iterations, it does not necessarily translate into memory bandwidth reduction since the adjacent data elements in the same cache line are being fetched despite being unused. Moreover, skipping computations reduces the time the processor works on a cache line, shortening the interval between cache line fetches and thus putting even more pressure on the memory subsystem. Since data subsetting utilizes a subset buffer, out-of-subset data elements are not fetched. This result underscores the effectiveness of data-centric approaches in the context of memory-bound applications.

5.6.3 Choice of Access Redirection Function

This subsection studies the impact of ARF choice on application quality (classification accuracy). To this end, this subsection considers 2 ARFs – periodic *vs.* neighborhood-average. In periodic subsetting, every K^{th} is selected to be part of the subset. Neighborhood-average also picks one in every K elements, but places the average of all elements in each interval of K in the subset buffer. Figure 5.7 shows the degradation in output quality as the degree of subsetting is increased from 1 to 4. In all cases, a degradation in quality is observed as the subsetting period is increased. However, neighborhood-average incurs

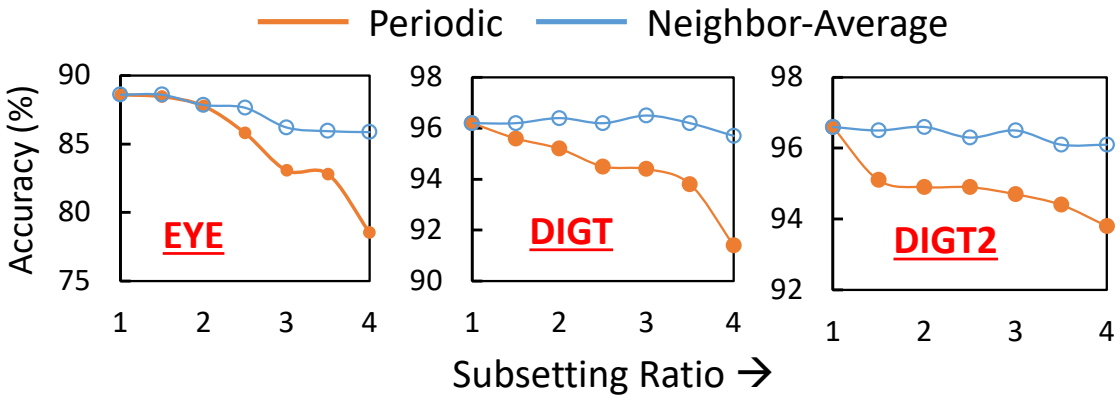


Figure 5.7. Impact of ARF choice on accuracy

very little quality degradation compared to periodic ARF since it accounts for out-of-subset elements. Therefore, it is useful for datasets that exhibit lower spatial locality.

5.6.4 Dynamic Modulation of Subset Size

A desirable feature of `SubsettableTensor` is the ability to modulate the subset size during execution. This can be leveraged in the context of iterative algorithms such as K-Means clustering (Section 5.4.2). This is demonstrated in action in Figure 5.8, which plots how the quality metric (average distance between data points and their cluster centroids) converges over time when iterations are begun with different sizes of subsets. It is shown that the original algorithm (blue line: $R = 1$) takes the longest time to converge since each iteration takes longer. When clustering starts on only half of the data elements which are randomly picked (orange line: $R = 2$), it is shown that the algorithm converges quicker, but converges to a sub-optimal solution at a higher average distance. At this point, the subset size is increased to include all elements in the data structure, which enables it to converge to the same average distance as the original algorithm, while achieving 1.62× performance improvement. A similar behavior is observed when the algorithm begins with even smaller subset sizes ($R = 3, 4$), but beyond a point no further performance improvements can be seen, as the subset sizes are too small to move the cluster centroids in the correct direction.

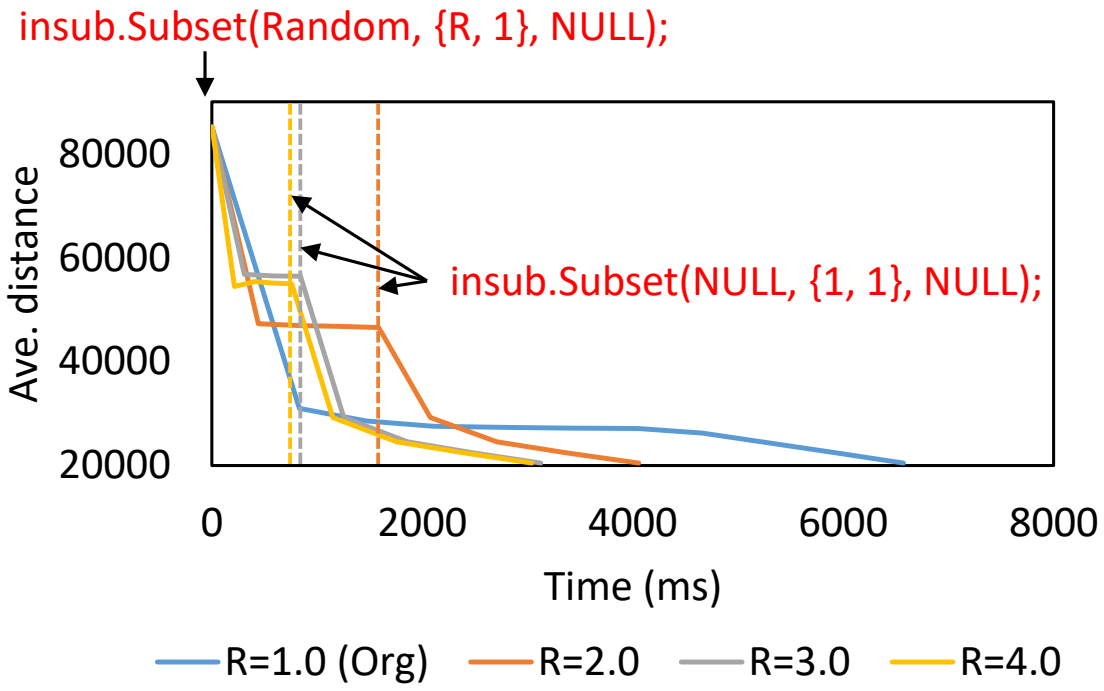


Figure 5.8. Data subsetting applied to K-Means clustering

5.7 Summary

Optimizing memory bandwidth is key to the efficiency of emerging data-intensive applications on modern computing platforms. This chapter addresses this challenge using approximate computing, wherein a data-centric approach is adopted. Specifically, this chapter proposes an approximation technique called data subsetting, which approximates accesses to a data structure by redirecting them to fall within a pre-defined subset of elements. This enhances performance as the foot-print of the application's memory accesses is significantly reduced. This chapter realizes data subsetting through a templated data structure called `SubsettableTensor`, which allows application software to specify and dynamically modify which elements constitute the accessible subset, and how access to other parts of the data structure are redirected. Additionally, this chapter proposes optimizations such as the use of a subset buffer, and identifying and skipping redundant computations to enhance the benefits of data subsetting on commercial systems. Finally, this chapter evaluates data subsetting

on parallel software implementations of 7 machine learning applications and demonstrate 1.33×-4.44× improvement in performance.

Optimizing memory bandwidth is key to the efficiency of emerging data-intensive applications on modern computing platforms. This chapter addresses this challenge using approximate computing, wherein a data-centric approach is adopted. Specifically, an approximation technique called data subsetting is proposed, which approximates accesses to a data structure by redirecting them to fall within a pre-defined subset of elements. This enhances performance as the foot-print of the application's memory accesses is significantly reduced. Data subsetting is realized through a templated data structure called `SubsettableTensor`, which allows application software to specify and dynamically modify which elements constitute the accessible subset, and how accesses to other parts of the data structure are redirected. Additionally, optimizations such as the use of a subset buffer, and identifying and skipping redundant computations to enhance the benefits of data subsetting are proposed. Data subsetting is evaluated on parallel software implementations of 7 machine learning applications and demonstrate 1.33×-4.44× improvement in performance.

6. CONCLUSION

Approximate computing has emerged as a new design paradigm for trading off performance/energy improvement with a tolerable quality degradation by taking advantage of the application's intrinsic resistance to internal computational errors. In order to expedite a wide-spread adoption of approximate computing by designers, this thesis proposes three techniques for approximate computing across the stack which requires a minimal change to the conventional design flow and are runtime controllable. First, *clock overgating* targets the Register Transfer Level (RTL) which is the most widely used level of abstraction for HW design. Clock overgating extends the duration of traditional clock gating to selected Flip-Flops (FFs) in the circuit so that dynamic power in the clock tree and their downstream logic can be saved while allowing chances for errors in the FF values to affect the final output. This thesis provides a systematic methodology for identifying an optimal clock overgating configuration based on the given RTL design and quality constraint. Experimental results show that the proposed methodology can produce more power-efficient approximate circuits compared to conventional precision scaling while searching through the exponentially large design space within a reasonable time using a gradient descent approach. Second, *Value Similarity eXtensions (VSX)* enhances general-purpose processors by exploiting value similarity among neighbor elements of data structures. VSX dynamically detects and skips instructions that are expected to produce similar result to a previous one. High performance benefits can be expected by avoiding such instructions from entering the processor pipeline. This thesis provides strategies for realizing the proposed approach such as identifying potentially skipped instructions from user annotation in SW, extracting similarity information from the data cache, and saving & reusing previous instruction results in place of a skipped instruction. In addition, multiple loop iterations that produce similar results are replaced by a separate group of instructions to enable further speedup. Experimental results show that the proposed micro-architectural extensions skip a large portion of loads, computes, and iterations while incurring a small area overhead. Finally, *data subsetting* offers a new way of approximation through restricting accesses to a subset of data, which can be applied to software design for commodity platforms while ensuring minimal deviation from

the conventional coding practice. By redirecting all data accesses to a subset of data, overall memory footprint is decreased and thus memory-intensive applications can benefit from reduced memory bandwidth. This thesis provides *SubsettableTensor*, a software template which guides programmers to identify an appropriate subset of data and perform access redirections in a cache-friendly manner. Experimental results show that the proposed approach outperforms conventional loop perforation by effectively reducing the memory bandwidth, and can be dynamically reconfigured in an advantageous way for certain type of applications such as iterative algorithms. In summary, the hardware and software techniques for approximate computing proposed in this thesis show an adequate potential for pushing approximate computing paradigm towards the mainstream design practice.

REFERENCES

[1] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Design Automation Conference*, Jun. 2015, pp. 1–6.

[2] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9.

[3] S. Venkataramani, "Approximate computing: An integrated cross-layer framework," PhD thesis, 2016, p. 198, ISBN: 9781369641103.

[4] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 728–735.

[5] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "IMPACT: Imprecise adders for low-power approximate computing," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug. 2011, pp. 409–414.

[6] N. Zhu *et al.*, "An enhanced low-power high-speed adder for error-tolerant application," in *Proc. ISIC*, 2009, pp. 69–72.

[7] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Design Automation Conference*, 2012, pp. 820–825.

[8] J. Huang *et al.*, "A methodology for energy-quality tradeoff using imprecise hardware," in *Design Automation Conference*, 2012, pp. 504–509.

[9] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*, Jan. 2011, pp. 346–351.

[10] P. K. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications," in *Design, Automation and Test in Europe*, Mar. 2011.

[11] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, San Diego, California, USA: ACM, 1999, pp. 30–35.

[12] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *2010 Design, Automation and Test in Europe*, Mar. 2010, pp. 957–960.

[13] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *2011 Design, Automation and Test in Europe*, Mar. 2011, pp. 1–6.

[14] A. Lingamneni, C. Enz, J. Nagel, K. Palem, and C. Piguet, "Energy parsimonious circuit design through probabilistic pruning," in *2011 Design, Automation and Test in Europe*, 2011, pp. 1–6.

[15] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Design Automation Conference 2012*, Jun. 2012, pp. 796–801.

[16] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 1–12.

[17] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 176–187.

[18] A. Raha and V. Raghunathan, "qLUT: Input-aware quantized table lookup for energy-efficient approximate accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017.

[19] M. Riera, J. Arnau, and A. Gonzalez, "Computation reuse in DNNs by exploiting input similarity," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 57–68.

[20] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 164–174.

[21] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2013, pp. 1–12.

[22] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[23] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 449–460.

[24] S. T. Chakradhar and A. Raghunathan, "Best-effort computing: Re-thinking parallel software and hardware," in *Design Automation Conference*, Jun. 2010.

[25] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Design Automation Conference*, 2010, pp. 555–560.

[26] D. Shin and S. K. Gupta, "A re-design technique for datapath modules in error tolerant applications," in *Proc. ATS*, Nov. 2008, pp. 431–437.

[27] L. N. Chakrapani *et al.*, "Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Atlanta, GA, USA: ACM, 2008, pp. 187–196.

[28] Z. Kedem, V. J. Mooney, K. K. Muntimadugu, K. V. Palem, A. Devarasetty, and P. D. Parasuramuni, "Optimizing energy to minimize errors in dataflow graphs using approximate adders," in *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '10, Scottsdale, Arizona, USA, 2010, pp. 177–186.

[29] K. Du, P. Varman, and K. Mohanram, "High performance reliable variable latency carry select addition," in *Design, Automation and Test in Europe*, 2012, pp. 1257–1262.

[30] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 48–54.

[31] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[32] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 825–831.

[33] S. Lee, L. K. John, and A. Gerstlauer, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *Design, Automation and Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 187–192.

[34] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *2011 Design, Automation and Test in Europe*, 2011, pp. 1–6.

[35] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *2013 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1367–1372.

[36] S. Jain, S. Venkataramani, and A. Raghunathan, "Approximation through logic isolation for the design of quality configurable circuits," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 612–617.

[37] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: Modeling and analysis of circuits for approximate computing," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 667–673.

[38] B. Zhai, S. Pant, L. Nazhandali, S. Hanson, J. Olson, A. Reeves, M. Minuth, R. Helfand, T. Austin, D. Sylvester, and D. Blaauw, "Energy-efficient subthreshold processor design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 8, pp. 1127–1137, 2009.

[39] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, Dresden, Germany: European Design and Automation Association, 2010, pp. 335–338.

[40] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *2010 Design, Automation and Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 1560–1565.

[41] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 505–516.

[42] A. Jain, P. Hill, S. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[43] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Exploiting staleness for approximating loads on cmps," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 343–354.

[44] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: A cache for approximate computing," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 50–61.

[45] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, "The bunker cache for spatio-value approximation," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[46] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *2013 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2013, pp. 202–207.

[47] P. Guo, B. Hu, R. Li, and W. Hu, "FoggyCache: Cross-device approximate computation reuse," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '18, New Delhi, India, 2018, pp. 19–34.

[48] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 554–566.

[49] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthsarathy, "Lazy pipelines: Enhancing quality in approximate computing," in *2016 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1381–1386.

[50] Y.-C. Hu, M. T. Lokhandwala, T. I., and H.-W. Tseng, "Dynamic multi-resolution data storage," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, 2019, pp. 196–210.

[51] K. Cho, Y. Lee, Y. H. Oh, G. Hwang, and J. W. Lee, "eDRAM-based tiered-reliability memory with applications to low-power frame buffers," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2014, pp. 333–338.

[52] P. Düben, Parishkrati, S. Yenugula, J. Augustine, K. Palem, J. Schlachter, C. Enz, and T. N. Palmer, "Opportunities for energy efficient computing: A study of inexact general purpose processors for high-performance and big-data applications," in *2015 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2015, pp. 764–769.

[53] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, "RRAM-based analog approximate computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, pp. 1905–1917, 2015.

[54] A. Rahimi, A. Ghofrani, K. Cheng, L. Benini, and R. K. Gupta, "Approximate associative memristive memory for energy-efficient GPUs," in *2015 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2015, pp. 1497–1502.

[55] H. Zhang, M. Putic, and J. Lach, "Low power GPGPU computation with imprecise hardware," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[56] I. J. Chang, D. Mohapatra, and K. Roy, "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.

[57] A. Ranjan, S. Venkataramani, Z. Pajouhi, R. Venkatesan, K. Roy, and A. Raghunathan, "STAxCache: An approximate, energy efficient STT-MRAM cache," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '17, Lausanne, Switzerland: European Design and Automation Association, 2017, pp. 356–361.

[58] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[59] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt, "Exploiting partially-forgetful memories for approximate computing," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 19–22, 2015.

[60] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 25–36.

[61] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[62] A. Raha, H. Jayakumar, and V. Raghunathan, "A power efficient video encoder using reconfigurable approximate arithmetic units," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, 2014, pp. 324–329.

[63] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *2015 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2015, pp. 665–670.

[64] T. M. Aamodt and P. Chow, "Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 26:1–26:27, May 2008.

[65] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini, "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2013, 35:1–35:10.

[66] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 51–66.

[67] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 198–209.

[68] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 35–50.

[69] Aurangzeb and R. Eigenmann, "HiPA: History-based piecewise approximation for functions," in *Proceedings of the International Conference on Supercomputing*, 2017, 23:1–23:10.

[70] J. S. Miguel and N. E. Jerger, "The anytime automaton," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 545–557.

[71] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[72] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible approximation with selective dynamic loop perforation," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 341–351.

[73] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 13–24.

[74] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing approximations to MapReduce frameworks," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 383–397.

[75] R. Akram, M. M. U. Alam, and A. Muzahid, "Approximate lock: Trading off accuracy for performance by skipping critical sections," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2016.

[76] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 2007.

[77] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 427–428.

[78] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing program semantics to unleash parallelization," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015, pp. 235–245.

[79] M. A. Breuer, "Multi-media applications and imprecise computation," in *Proc. Euromicro Conf. on Digital System Design*, 2005, pp. 2–7.

[80] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of approximate sequential circuits," in *2014 Design, Automation and Test in Europe Conference Exhibition (DATE)*, Mar. 2014, pp. 1–6.

[81] Chaofan Li, Wei Luo, S. S. Sapatnekar, and Jiang Hu, "Joint precision optimization and high level synthesis for approximate computing," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[82] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *2014 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–6.

[83] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language support for approximate hardware design," in *2015 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2015, pp. 812–817.

[84] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, Mar. 2016.

[85] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, 2016.

[86] Y. Kim, S. Venkataramani, K. Roy, and A. Raghunathan, "Designing approximate circuits using clock overgating," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16, Austin, Texas: Association for Computing Machinery, 2016.

[87] C. Wu *et al.*, "Machine learning at Facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 331–344.

[88] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.

[89] Y. Sato, T. Tsumura, T. Tsumura, and Y. Nakashima, "An approximate computing stack based on computation reuse," in *2015 Third International Symposium on Computing and Networking (CANDAR)*, 2015, pp. 378–384.

[90] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 301–312.

[91] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 127–139.

[92] X. He, G. Yan, Y. Han, and X. Li, "ACR: Enabling computation reuse for approximate computing," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 643–648.

[93] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, 62:1–62:26, Jan. 2016.

[94] Y. Kim, S. Venkataramani, N. Chandrachoodan, and A. Raghunathan, "Data subsetting: A data-centric approach to approximate computing," in *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2019, pp. 576–581.

[95] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "SparCE: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 912–925, Jun. 2019.

[96] HP Labs. (2021). CACTI, [Online]. Available: https://www.hpl.hp.com/research/cacti/.

[97] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[98] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, pp. 675–678.

[99] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *Proc. IPDPS*, 2009, pp. 1–12.

[100] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 213–224.

[101] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate DRAM," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[102] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan, "Approximate memory compression for energy-efficiency," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Jul. 2017, pp. 1–6.

[103] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system," in *Design Automation Conference*, 2017, 74:1–74:6.

[104] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan, "Quality configurable approximate DRAM," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1172–1187, Jul. 2017.

VITA

Younghoon Kim received the bachelor's degree in Electrical and Computer Engineering and the master's degree in Electrical Engineering and Computer Science from Seoul National University, Seoul, South Korea in 2008 and 2010. He is currently pursuing a Ph.D. degree in the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA. After his graduation, Younghoon will join Qualcomm, San Diego, CA, USA as an SoC architect. Previously, Younghoon was a firmware engineer at COWON Systems, a portable multimedia player manufacturer in Seoul, South Korea from 2010 to 2013. He has also interned with the Data Center Group at Intel, Hillsboro, OR, USA during the Summer of 2017. His primary research interests include approximate computing and high-performance and low-power system design. Younghoon received the National Science and Engineering Undergraduate Scholarship from Korea Student Aid Foundation. His research on clock overgating has received a best paper nomination from DAC 2016, and his research on data subsetting has received the best paper award from DATE 2019.