Architectural Exploration of Data Recomputation for Improving Energy Efficiency

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

Ismail Akturk

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Ulya R. Karpuzcu

July, 2017

© Ismail Akturk 2017
ALL RIGHTS RESERVED

Acknowledgements

I would like to thank my advisor Ulya R. Karpuzcu for her continued support, advice, and guidance through the years. She was always ready to help, and she never lost her patience and never gave up on me to make me a better researcher. I could never thank her enough.

I am grateful to my final exam committee members David Lilja, Sachin Sapatnekar, and Pen-Chung Yew for their support and time.

Furthermore, I would like to thank my colleagues in the department, existing and former members of the ALTAI, and folks in 4-166. They made my time better by their friendship; special thanks to Karen and Hari.

Finally, I would like to express my heartfelt gratitude to my family for their patience and support, especially to Aysegul.

Dedication

All praise is due to God alone, the Sustainer of all the worlds, the Most Gracious, the Dispenser of Grace.

Abstract

There are two fundamental challenges for modern computer system design. The first one is accommodating the increasing demand for performance in a tight power budget. The second one is ensuring correct progress despite the increasing possibility of faults that may occur in the system.

To address the first challenge, it is essential to track where the power goes. The energy consumption of data orchestration (i.e., storage, movement, communication) dominates the energy consumption of actual data production, i.e., computation. Oftentimes, recomputing data becomes more energy efficient than storing and retrieving pre-computed data by minimizing the prevalent power and performance overhead of data storage, retrieval, and communication. At the same time, recomputation can reduce the demand for communication bandwidth and shrink the memory footprint. In the first half of the dissertation, the potential of data recomputation in improving energy efficiency is quantified and a practical recomputation framework is introduced to trade computation for communication.

To address the second challenge, it is needed to provide scalable checkpointing and recovery mechanisms. The traditional method to recover from a fault is to periodically checkpoint the state of the machine. Periodic checkpointing of the machine state makes rollback and restart of execution from a safe state possible upon detection of a fault. The energy overhead of checkpointing, however, as incurred by storage and communication of the machine state grows with the frequency of checkpointing. Amortizing this overhead becomes especially challenging, considering the growth of expected error rates as an artifact of contemporary technology scaling. Recomputation of data (which otherwise would be read from a checkpoint) can reduce both the frequency of checkpointing, the size of the checkpoints and thereby mitigate checkpointing overhead. In the second half, quantitative characterization of recomputation-enabled checkpointing (based on recomputation framework) is provided.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Structure of The Dissertation 4

2 Motivation **5**

3 Amnesiac: Proof-of-Concept Framework for Recomputation **8**

3.1 Introduction 8

3.2 Amnesic Execution Semantics 9

3.2.1 Recomputation Slice (*RSlice*) 10

3.2.2 Non-recomputable Inputs 11

3.2.3 Side Effects 12

3.3 An Illustrative Proof-Of-Concept Amnesic Implementation 12

3.3.1 Amnesic Compiler and Instruction Set Extensions 13

3.3.2 Amnesic Microarchitecture 15

3.3.3 Amnesic Scheduler 16

3.3.4 Putting It All Together 17

3.3.5 Storage Complexity . . . 18
3.3.6 Technicalities . . . 19
3.4 Evaluation Setup . . . 20
3.5 Evaluation . . . 22
3.5.1 Impact on Energy Efficiency . . . 22
3.5.2 Impact on Instruction Count and Mix . . . 26
3.5.3 Memory Access Characteristics . . . 27
3.5.4 *RSlice* Characteristics . . . 30
3.5.5 Break-even Point . . . 31
3.5.6 Data Locality Analysis . . . 34
3.6 Related Work . . . 34

4 Recomputation Taxonomy 37
4.1 Introduction . . . 37
4.2 Recomputation Taxonomy . . . 38
4.2.1 Recalculation Based Recomputation . . . 39
4.2.2 Prediction Based Recomputation . . . 40
4.2.3 Recalculation + Prediction Based Recomputation . . . 40
4.3 Evaluation Setup . . . 41
4.4 Evaluation . . . 42
4.4.1 Impact on Energy and Performance . . . 42
4.4.2 Impact on Execution Semantics . . . 46
4.5 Summary . . . 50

5 Recomputation-enabled Checkpointing and Recovery 51
5.1 Introduction . . . 51
5.2 Recomputation: Basic Idea . . . 53
5.2.1 Support for Recomputation . . . 53
5.2.2 Recap: Recomputation Framework . . . 54
5.3 Checkpointing and Recovery . . . 55
5.3.1 Checkpointing . . . 55
5.3.2 Error Detection and Recovery . . . 56
5.4 Incorporating Recomputation in Checkpointing and Recovery . . . 57

5.4.1 Recomputation Enabled Checkpointing 57
5.4.2 Recomputation Enabled Recovery 58
5.4.3 Microarchitecture Support for Recomputation-Enabled Checkpointing 58
5.4.4 Overheads 60
5.5 Evaluation Setup 61
5.6 Evaluation 63
5.6.1 Checkpointing Overhead in Fault-Free Execution 63
5.6.2 Recovery Overhead in Fault-Occurring Execution 65
5.6.3 Checkpoint and Footprint Size Reduction 68
5.6.4 Impact of Thread Count on Checkpointing Overhead 69
5.6.5 Impact of Fault Rate on Recovery Overhead 72
5.6.6 Impact of Checkpoint Frequency on Checkpointing Overhead . . 74
5.6.7 Coordinated Local vs. Global Checkpointing 77
5.6.8 Impact of *RSlice* Length on Checkpoint Size 82
5.7 Related Work 84

6 Conclusion **87**

References **89**

Appendix A. Impact of *RSlice* Length on Checkpoint Size **97**

List of Tables

2.1 Strong vs. weak scaling for an n-fold increase in core count. Best case scenario, excluding communication overhead. PS: problem size. 6
2.2 Energy consumption of 64-bit computation and communication adapted from [6]. 7
3.1 Benchmarks deployed to quantify the potential of amnesic execution. . 21
3.2 Simulated architecture to quantify the potential of amnesic execution. . 22
3.3 Dynamic instruction mix and energy breakdown under amnesic execution. 27
3.4 Memory access profile of load instructions under classic execution, which are swapped for recomputation under *Compiler*, *FLC*, and *LLC* policies, respectively. 28
3.5 Break-even point (for *C-Oracle*). 32
4.1 Benchmarks deployed to quantify the potential of different recomputation techniques. 42
5.1 Simulated architecture to evaluate the impact of recomputation on checkpointing and recovery. 61
5.2 The summary of configurations evaluated. 63

List of Figures

1.1 Microscopic view per machine state transition. 2
1.2 Execution semantic under recomputation. 3
3.1 Example Recomputation Slice, *RSlice(v)*. 10
3.2 Amnesic Microarchitecture & Scheduler. 15
3.3 EDP gain under amnesic execution. 23
3.4 Energy gain under amnesic execution. 25
3.5 % reduction in execution time. 26
3.6 Histograms of instruction count per recomputed *RSlice* under *Compiler* policy. 29
3.7 % of *RSlice*s with **n**on-re**c**omputable leaf inputs. 31
3.8 % value locality of loads (under classic execution), which are swapped for recomputation by the *Compiler* policy. 33
4.1 Classic execution vs. Recomputation. 37
4.2 Energy gain under recomputation. 43
4.3 Performance gain under recomputation. 44
4.4 EDP gain under recomputation. 44
4.5 EDP gain under **prediction** as a function of value locality threshold for prediction. 45
4.6 EDP gain under **recalculation+prediction** as a function of value locality threshold for prediction. 46
4.7 Value locality of *RSlice* instructions. 47
4.8 Node count of *RSlice*s before (**recalculation**) and after pruning (**recalculation+prediction**). 49
5.1 Microarchitectural support needed to facilitate recomputation. 54

5.2 Recovery from a fault. 56
5.3 Normalized execution time of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_NF* and *Rec_Ckpt_NF* configurations. 64
5.4 Normalized energy consumption of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_NF* and *Rec_Ckpt_NF* configurations. 64
5.5 EDP reduction of benchmarks under *Rec_Ckpt_NF* configuration (w.r.t. *Ckpt_NF*). 65
5.6 Normalized execution time of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_F* and *Rec_Ckpt_F* configurations. 66
5.7 Normalized energy consumption of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_F* and *Rec_Ckpt_F* configurations. 67
5.8 EDP reduction of benchmarks under *Rec_Ckpt_F* configuration (w.r.t. *Ckpt_F*). 67
5.9 Percentage of checkpoint size reduction under *Rec_Ckpt_NF* configuration. 69
5.10 Percentage of footprint size reduction under *Rec_Ckpt_NF* configuration. 70
5.11 Performance overhead of checkpointing for 8- 16- and 32-threaded executions under *Ckpt_NF* configuration. 71
5.12 Performance overhead reduction of checkpointing for benchmarks running with 8-, 16-, and 32-threads under *Rec_Ckpt_NF* configuration. 72
5.13 Normalized execution time under *Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates. 73
5.14 Normalized execution time under *Rec_Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates. 74
5.15 Normalized EDP under *Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates. 75
5.16 Normalized execution time under *Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies. 76
5.17 Normalized execution time under *Rec_Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies. 77
5.18 Normalized EDP under *Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies. 77

5.19 Normalized execution time of *Ckpt_NF* and *Rec_Ckpt_NF* for coordinated local checkpointing (w.r.t. *Ckpt_NF* and *Rec_Ckpt_NF* for global checkpointing, respectively). 78
5.20 Normalized EDP of *Ckpt_NF* and *Rec_Ckpt_NF* for coordinated local checkpointing (w.r.t. *Ckpt_NF* and *Rec_Ckpt_NF* for global checkpointing, respectively). 79
5.21 Normalized execution time of *Ckpt_F* and *Rec_Ckpt_F* for coordinated local checkpointing (w.r.t. *Ckpt_F* and *Rec_Ckpt_F* for global checkpointing, respectively). 81
5.22 Normalized EDP of *Ckpt_F* and *Rec_Ckpt_F* for coordinated local checkpointing (w.r.t. *Ckpt_F* and *Rec_Ckpt_F* for global checkpointing, respectively). 82
5.23 Total checkpoint size reduction as a function of *RSlice* length for bt. . . 83
5.24 Impact of *RSlice* length on checkpoint size over time for bt. 84
A.1 Total checkpoint size reduction as a function of *RSlice* length for cg. . . 97
A.2 Total checkpoint size reduction as a function of *RSlice* length for dc. . . 98
A.3 Total checkpoint size reduction as a function of *RSlice* length for ft. . . 98
A.4 Total checkpoint size reduction as a function of *RSlice* length for is. . . 98
A.5 Total checkpoint size reduction as a function of *RSlice* length for lu. . . 99
A.6 Total checkpoint size reduction as a function of *RSlice* length for mg. . . 99
A.7 Total checkpoint size reduction as a function of *RSlice* length for sp. . . 99

Chapter 1

Introduction

Under contemporary scaling, a given chip area can still accommodate more compute engines (in the form of general-purpose cores or accelerators) each technology generation. However, cooling and power delivery limitations prevent a proportional expansion of the power budget. As a result, we can simultaneously utilize only a progressively diminishing fraction of on-chip resources, and the rest has to stay un-powered, aka *dark* [1, 2]. To illuminate dark silicon, we need to carefully track where the power goes among the components of the chip. The data and control flow throughout the execution of a program trigger a sequence of machine state transitions. As depicted in Figure 1.1, each state transition encompasses the following tasks:

- retrieval of input state (i.e., inputting)
- compute output state from inputs (i.e., processing)
- write output state (i.e., outputting)
- hold new machine state (i.e., storing)

These tasks are carried over six basic steps: Upon retrieval of input state (i.e., (1) & (2)), compute engines derive output state from inputs (i.e., (3)). Next comes storage of output state (i.e., (4) & (5)) and retention of new machine state (i.e., (6)) until the next transition.

Power goes to all of these steps, with the actual computation (i.e., (3)) representing the least energy-hungry [3, 4].

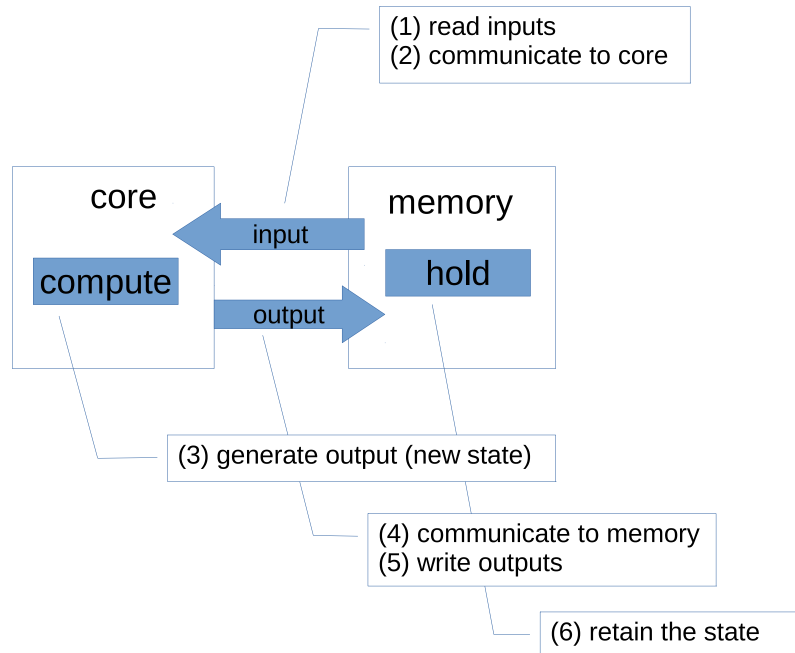

Figure 1.1: Microscopic view per machine state transition.

The building blocks of classic computing, transistors, consume dynamic power as they toggle and static power due to leakage when turned off (because of restrictions from technology scaling). Typically only a subset of transistors toggle during a state transition, therefore, dynamic-power-heavy steps such as (3) can also consume static power. On the other hand, static-power-heavy steps such as (6) also consume dynamic power due to control logic. The breakdown of total power consumption across steps, and the ratio of dynamic to static power per step evolve as a function of the operating regime and technology.

Unfortunately, emerging technology solutions are not mature enough to meet the growing capacity, bandwidth, and performance demand with-in the stringent power budget. Imbalances between logic and memory technologies further result in rising time and power, hence energy (time $\times$ power) expenditure in steps (1), (2), (4) and (5) (along with (6) depending on the memory technology) [3, 4]. As a consequence, reproducing, i.e., recomputing data oftentimes becomes more energy efficient than storing

and retrieving pre-computed data.

Data recomputation replaces the load of inputs with the reproduction of the input data. Step (1) incurs the time and power overhead of the memory access to perform the read; and (2) incurs the time and power overhead of the subsequent communication of inputs to the compute engines. Recomputation transforms the overhead of (1) & (2) to the overhead of the recomputation of inputs. The energy savings comes from (1) & (2) being much more energy-hungry than computation (i.e., (3)).

Replacing loads with recomputation may unlock further opportunities for energy savings: Each input represents the output of a previous step in execution. In other words, each consumer load has a matching producer store. For each load replaced with recomputation, the corresponding store (to the same memory address) can become redundant if no other load (from the same address) depends on it. Therefore, recomputation can also filter out output stores and cut off the time and power overhead of (4) & (5). Step (4) incurs the overhead of communication of outputs to memory; and (5) incurs the overhead of the subsequent memory access to perform the write.

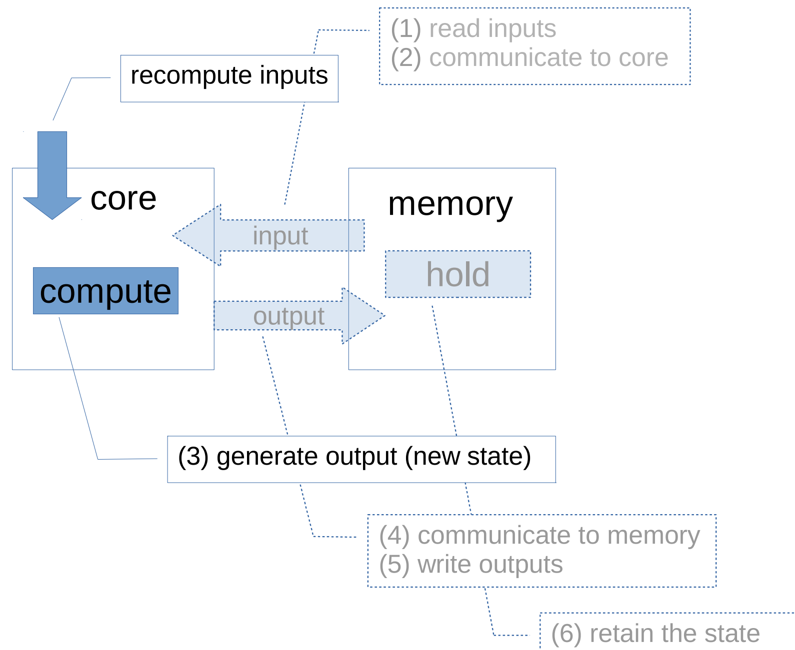

Figure 1.2: Execution semantic under recomputation.

Putting it all together, data recomputation can improve energy efficiency by

- Replacing input loads with recomputation of inputs, hence transforming (1) & (2) into the less energy-hungry (3).
- Filtering out stores which represent the producers of the loads replaced with recomputation, hence cutting off the overhead of (4) & (5) (along with (6) depending on the memory technology).

1.1 Structure of The Dissertation

The rest of the dissertation is organized as follows:

- Chapter 2 presents the motivation behind the data recomputation and its potential to improve energy efficiency.
- Chapter 3 illustrates a proof-of-concept recomputation framework and provides quantitative characterization.
- Chapter 4 explores different forms of recomputation and provides a recomputation taxonomy.
- Chapter 5 introduces recomputation-enabled checkpointing, and provides quantitative characterization.
- Chapter 6 summarizes our contribution and concludes the discussion.

Chapter 2

Motivation

In general, the communication of data can be categorized into two. The first one is *vertical* communication where data is communicated to the compute engine through local memory hierarchy. The data retrievals performed by sequential applications are examples of this kind of communication. It can also be referred as intra-core communication. The second one is *horizontal* communication where data generated by a compute engine is communicated to the other compute engines through memory (in case of shared-memory system), or through off-chip interconnection network (in case of distributed memory system). Since data communication takes place across the compute engine boundaries, this type of communication is called horizontal communication. It can also be referred as inter-core communication. Regardless of type, communication energy dominates the energy used for actual data production, i.e. computation. Therefore, oftentimes, recomputing data becomes more energy efficient than communicating data in both horizontal (i.e., inter-core) and vertical (i.e., intra-core) directions.

The magnitude and the frequency of inter-core communication depends on how the problem being solved distributes the data among the cores. Problem size dictates the total amount of data processed across all cores. As more cores become available, the problem can scale in two distinct ways to translate the increase in core count into enhanced performance (as measured by the total amount of data processed over the overall processing time): *strong scaling* or *weak scaling*.

Table 2.1 captures how the total and per core problem size (*PS*), execution time (t), and throughput performance (*PS/t*) evolve for an n-fold increase in core count. Under

Scaling	(Total) PS	PS per core	time (t)	PS/t	PS share (per core)
Strong	-	/n	/n	×n	/n
Weak	×n	-	-	×n	/n

Table 2.1: Strong vs. weak scaling for an n-fold increase in core count. Best case scenario, excluding communication overhead. PS: problem size.

strong scaling, the overall problem size, the total amount of data processed, remains constant. Each core processes progressively smaller chunk of data as the core count increases (*PS per core* decreases by n×), and in return finishes earlier. As a result, *PS/t* increases by $n\times$. The share of the problem per core reduces proportionally to the (increase in) core count.

On the other hand, under *weak scaling* [5], the problem size *per core* (thus the amount of data processed *per core*) remains constant which renders no change in the per core processing time (which dictates the overall processing time) as the core count increases. At the same time, the overall problem size (the total amount of data processed across all cores), grows proportionally to the (increase in) core count (increases by n×). Therefore, each core processes a progressively smaller fraction of the total amount of data as the core count increases as tabulated in the last column. The share of the problem per core still reduces proportionally to the (increase in) core count.

Under both scaling scenarios, higher levels of concurrency imply a lower fraction of the total amount of data in close physical proximity to each core, which hurts *data locality*, and increases the likelihood of more frequent communication. As concurrency hurts data locality, each core must spend both more time and power in communication. Consequently, communication energy, as induced by data movement and the orchestration thereof, is expected to dominate computation energy [3].

Emerging non-volatile memories can minimize hold energy due to the premise of (practically) zero static power, but suffer from excessive write energy. Thus memory energy would still dominate computation energy.

Table 2.2 adapted from [6], shows how communication energy, as characterized by a 64-bit data transfer across chip, changes as technology scales. Communication energy increases from 1.55× computation energy at 40nm to approximately 6× at 10nm

Process Technology	40nm	10nm	
Operating Voltage	0.9V	0.75V (HP)	0.65V (LP)
64-bit double precision FLOP	50pJ	8.7pJ	6.5pJ
64-bit transfer on chip (10mm)	77.5pJ (1.55x)	50.02pJ (5.75x)	37.5pJ (5.77x)

Table 2.2: Energy consumption of 64-bit computation and communication adapted from [6].

(considering processes optimized for high performance, HP, and low power, LP). Since communication energy tends to grow with distance, a similar trend applies for off-chip communication. Therefore, communication energy becomes even more prominent with technology scaling.

3D Stacking, or emerging photonics based interconnects, can render a lower off-chip (and potentially on-chip) communication energy when compared to state-of-the-art, but would not alter the communication-centric nature of parallel processing: Engaging more cores into computation reduces per core work, therefore, the mean time to communication, orthogonal to the technology of the communication medium. Accordingly, communication would still be the most energy-hungry phase.

As a consequence, recomputing data can become more energy-efficient than storing and retrieving pre-computed data. In this dissertation, we hence investigate the effectiveness of recomputing data values in minimizing, if not eliminating, the overhead of expensive off-chip memory accesses.

Chapter 3

Amnesiac: Proof-of-Concept Framework for Recomputation

3.1 Introduction

In this chapter, we investigate the effectiveness of recomputing data values in minimizing, if not eliminating, the overhead of expensive off-chip memory accesses. The idea is replacing a load with a sequence of instructions to recompute the respective data value, only if it is more energy-efficient. We call the resulting execution model *amnesic* [1] to contrast recomputation with conventional, *classic* execution.

Whether recomputation of a data value v can improve the energy efficiency or not tightly depends on where in the memory hierarchy the corresponding load would be serviced under classic execution, i.e., where in the memory hierarchy v resides. This is because the location of v in the memory hierarchy dictates the energy consumption of the respective load, $E_{ld,v}$, which in turn sets the energy budget for recomputation. Recomputation of v itself incurs an energy cost, $E_{rc,v}$, due to the (re)execution of the sequence of instructions to generate v. We will refer to each instruction in such a sequence as a *recomputing* instruction. Therefore, unless $E_{ld,v}$ exceeds $E_{rc,v}$, amnesic execution cannot improve energy efficiency.

Under amnesic execution, the sequence of recomputing instructions to generate v

[1] amnesia [am'nēZHə]: noun, a partial or total loss of memory.
amnesiac [am'nēzēˌak], amnesic [-zik, -sik]: noun & adjective.

form a backward slice, which we will refer to as *recomputation slice*, *RSlice*. The first instruction in the slice is the immediate producer of v, $P(v)$. To be able to (re)execute $P(v)$, each input operand of $P(v)$ should be readily available at the anticipated time of recomputation. This may not always be the case, and (re)execution of $P(v)$ may trigger the re(execution) of producers of $P(v)$'s input operands, recursively.

The recomputation slice to generate v, *RSlice(v)*, can grow by tracking producer-consumer dependencies for recomputing instructions, however, not indefinitely. First of all, the energy cost of recomputation of v, $E_{rc,v}$, increases with the number of recomputing instructions in *RSlice(v)*, and amnesic execution cannot be energy-efficient if $E_{rc,v}$ exceeds the energy consumption of the respective load, $E_{ld,v}$. At the same time, not all of the input operands of recomputing instructions can be (re)generated by recomputation. This may be the case if input operands correspond to (i) read-only values to be loaded from memory, such as program inputs; or (ii) register values which are lost, i.e., overwritten at the time of recomputation.

Swapping loads for recomputation slices can reduce the pressure on memory bandwidth and unlock further opportunities for energy savings: For each load replaced with an *RSlice*, the corresponding store (to the same memory address) can become redundant if no other load (from the same address) depends on it. Therefore, amnesic execution can also filter out energy-hungry stores, and reduce the pressure on memory capacity by shrinking the memory footprint.

Under amnesic execution, the workload becomes more compute-intensive to make a better use of classic processors optimized for computation, as opposed to communication. In the following, we quantitatively characterize the energy efficiency potential of amnesic execution.

3.2 Amnesic Execution Semantics

Under amnesic execution, an energy-hungry load is swapped with a sequence of recomputing instructions, which form a recomputation slice, *RSlice*, iff the energy cost of recomputation along the *RSlice* remains below the energy consumption of the respective load. In other words, the energy consumption of the load sets the energy budget for recomputation along the *RSlice*. If the anticipated energy cost of recomputation

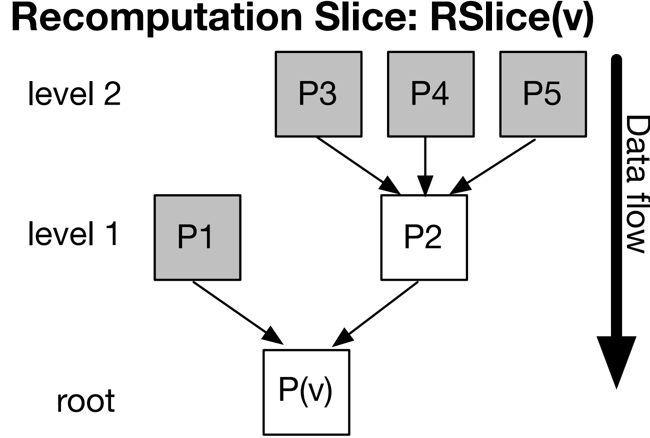

Figure 3.1: Example Recomputation Slice, *RSlice(v)*.

exceeds this budget, the respective load is performed and amnesic execution becomes equivalent to classic execution.

3.2.1 Recomputation Slice (*RSlice*)

For each data value v to be recomputed under amnesic execution, data dependencies determine the order of the recomputing instructions in *RSlice(v)*. *RSlice(v)* includes the immediate producer instruction of v, $P(v)$, and possibly, producer instructions of the input operands of $P(v)$, in a recursive manner. Producer instructions may come from different basic blocks or functions.

Recomputation slices are very unlikely to comprise all producer instructions (i.e., producers of the producers) along a dependency chain, as the energy cost of recomputation along an *RSlice* increases with the number of recomputing instructions, and can easily exceed the energy consumption of the respective load. Amnesic execution prohibits recomputation in this case.

Each recomputation slice, *RSlice(v)*, can be regarded as an upside-down tree with $P(v)$ residing at the root. Each node represents a producer instruction to be (re)executed. During recomputation along *RSlice(v)*, data flows from the leaves to the root. Figure 3.1 demonstrates an example. Nodes at level 1 correspond to immediate producers of the (input operands of the) root, nodes at level l correspond to the producers of nodes at

level *l-1*. The number of incoming branches at each node reflects the number of producers of the node. Hence, *RSlice(v)* is not necessarily a balanced tree. As (re)executing only a finite number of nodes can fit into the energy budget set by $E_{ld,v}$, *RSlice(v)* cannot grow indefinitely. At the same time, the energy cost of recomputation along *RSlice(v)* includes the cost of retrieving input operands of the leaf nodes (which cannot rely on producers to recompute their inputs).

In the example from Figure 3.1, P1 and P2 at level 1 correspond to producers of $P(v)$'s input operands. (Re)execution of P1 does not require any more (re)execution. (Re)execution of P2, on the other hand, requires the (re)execution of three of P2's producers: P3, P4, and P5, respectively. The leaf producers are all shaded in gray. The leaves either represent terminal instructions which do not have any producers (e.g., instructions with constants as input operands), or instructions for which (re)execution of their producers is not energy-efficient. Amnesic execution can only function, if the input operands of leaf instructions are available at their anticipated time of (re)execution.

3.2.2 Non-recomputable Inputs

Not all of the input operands of leaf instructions of an *RSlice* can be (re)generated by recomputation. This may be the case if input operands correspond to (i) read-only values to be loaded from memory, such as program inputs; or (ii) register values which are lost, i.e., overwritten at the time of recomputation. We will refer to such input operands as *non-recomputable* inputs. For amnesic execution to work, non-recomputable inputs of *RSlice* leaves should not only be available at the anticipated time of recomputation, but also be retrievable in an energy-efficient manner. Recomputation cannot eliminate any memory access to retrieve the non-recomputable inputs of *RSlice* leaves. If non-recomputable inputs do not reside in close physical proximity to the processor, the energy cost of their retrieval may easily exceed $E_{ld,v}$, rendering recomputation useless. In Section 3.3.2, we discuss dedicated buffering for non-recomputable inputs. No dedicated buffering is necessary if the leaf input operands correspond to constants or live register values.

3.2.3 Side Effects

For the discussion in this chapter, we focus on single-threaded amnesic execution[2] . Therefore, within the course of execution, recomputation along only one *RSlice* can be performed at a time. Amnesic execution should prevent corruption of the architectural state during recomputation, which can be achieved by allocating dedicated buffers (Section 3.3.2) similar to classic microarchitectural storage for speculative state.

Amnesic execution can orchestrate exception handling similar to exception handling under speculation, as well: record exceptions as long as recomputation along an *RSlice* is taking place, and defer their handling after recomputation finishes. However, we may need to revisit the definition of (im)precise exceptions in this case, since recomputation modifies the architectural control flow by executing extra (recomputing) instructions, as opposed to speculation.

3.3 An Illustrative Proof-Of-Concept Amnesic Implementation

The critical question under amnesic execution is *when to fire recomputation*. Potentially, the compiler can extract *RSlice(v)* for each load (to read v), by tracking data dependencies. Whether recomputation along *RSlice(v)* is more energy-efficient than performing the respective load, however, depends on where in the memory hierarchy v resides. Being able to only speculate where v can reside during execution, the compiler can at most probabilistically estimate the energy consumption of the respective load, $E_{ld,v}$, which sets the energy budget for recomputation. For each v where recomputation is estimated to be more energy-efficient, the compiler can modify the binary to swap the load for *RSlice(v)*. In the following, we will discuss various implementation options and how microarchitectural support can help.

[2] Under parallel execution, communication with memory expands along two dimensions: accesses to thread-local data and accesses to shared data. In this chapter, we focus on the first, in the context of single-threaded execution. In principle, loads swapped for recomputation may be triggered by core-to/from-memory (thread-local) or core-to-core (shared) communication.

The basic proof-of-concept implementation covered in this section features an amnesic compiler (Section 3.3.1), microarchitectural support for amnesic execution (Section 3.3.2), and a runtime (instruction) scheduler to orchestrate amnesic execution (Section 3.3.3). We first let the compiler identify and annotate a set of independent recomputation slices. Then, at runtime, the amnesic scheduler fires or skips recomputation along each *RSlice(v)*, by tracking where in the memory hierarchy v resides at the anticipated time of recomputation.

3.3.1 Amnesic Compiler and Instruction Set Extensions

The amnesic compiler first extracts a set of independent *RSlice*s as potential targets for recomputation, and annotates each, such that the amnesic scheduler (see Section 3.3.3) can identify them at runtime. The amnesic scheduler triggers recomputation along any given *RSlice(v)* only if loading the data value v is more energy-hungry than recomputation.

Slice Formation

The amnesic compiler pass first estimates, probabilistically (as detailed in the following and Section 3.4), the energy consumption of loading v, $E_{ld,v}$. Next comes dependency analysis to identify the producer instructions of v, in order to calculate the anticipated cost of potential recomputation. This step starts building *RSlice(v)* (where the immediate producer of v, $P(v)$, resides at the root), and lets *RSlice(v)* grow level by level, as long as the cumulative cost of recomputation along *RSlice(v)* being constructed remains below $E_{ld,v}$.

As the compiler traverses the dependency chains in constructing *RSlice(v)*, it may hit load instructions. In the proof-of-concept implementation, the compiler replaces each such load with the respective recomputing slice, recursively. Therefore, loads and stores cannot be present as intermediate nodes in *RSlice(v)*.

To derive the energy cost of recomputation, $E_{rc,v}$, the compiler pass uses instruction mix and count within *RSlice(v)*, along with machine specific energy per instruction (EPI) estimates: $E_{rc,v}$ is the sum of [*instruction count per category*] $\times$ [*EPI per category*], over all instruction categories represented in *RSlice(v)*'s instruction mix. $E_{ld,v}$ calculation, on the other hand, relies on probabilistic estimates: $\Pr_{Li}$, the probability of having a

load serviced by level Li in the memory hierarchy, is derived from hit and miss statistics of Li under profiling. Let the EPI estimate for a load serviced in Li be EPI_{Li}. Then, the sum of $\text{Pr}_{Li}\times$ EPI_{Li} over all levels i in the memory hierarchy (including off-chip) gives the probabilistic energy cost per load.

Slice Annotation

As a hint for the amnesic scheduler, the compiler replaces each load, the swap of which with recomputation is likely to be more energy-efficient (according to the probabilistic energy cost comparison explained above) with a special control flow instruction, `RCMP`. In this case, the compiler also inserts the constructed *RSlice* in the binary.

Semantically, `RCMP` corresponds to the fusion of a conditional branch with a load[3] . The resolution of the branching condition is left to the amnesic scheduler (see Section 3.3.3) at runtime. Depending on the branching condition (which is dictated by where in the memory hierarchy v resides at runtime), `RCMP` can act either as a branch to the entry point (starting from the leaves) of *RSlice(v)*, or as a classic load which reads v from memory. The latter is the case if the amnesic scheduler determines at runtime that recomputation is less energy-efficient than performing the load, i.e., $E_{rc,v}$ exceeds $E_{ld,v}$. Accordingly, as input operands, `RCMP` inherits all input operands of the respective load, in addition to the starting address of *RSlice(v)*.

At the exit of each such *RSlice(v)* embedded in the binary resides a return instruction, `RTN`, which returns the control to the instruction following `RCMP` in program order after recomputation along *RSlice(v)* finishes. `RTN` semantics closely mimic procedure return instructions. Before return, the recomputed data value v gets copied into the destination register of the eliminated load (recall that `RCMP` inherits all source and destination parameters of the respective load).

Only if the leaves of RSlice(v) have non-recomputable input operands, the compiler places `REC` instructions into the binary, which serve buffering of non-recomputable input operands such as overwritten register values. An `REC` instruction goes right after each instruction, a replica of which serves as a leaf in *RSlice(v)*. `REC` has a single integer operand: `leaf-address` which points to the address of the respective leaf instruction

[3] Depending on the specifics of the underlying instruction set architecture (ISA), `RCMP` can also be synthesized by a pair of branch and load instructions, without loss of generality.

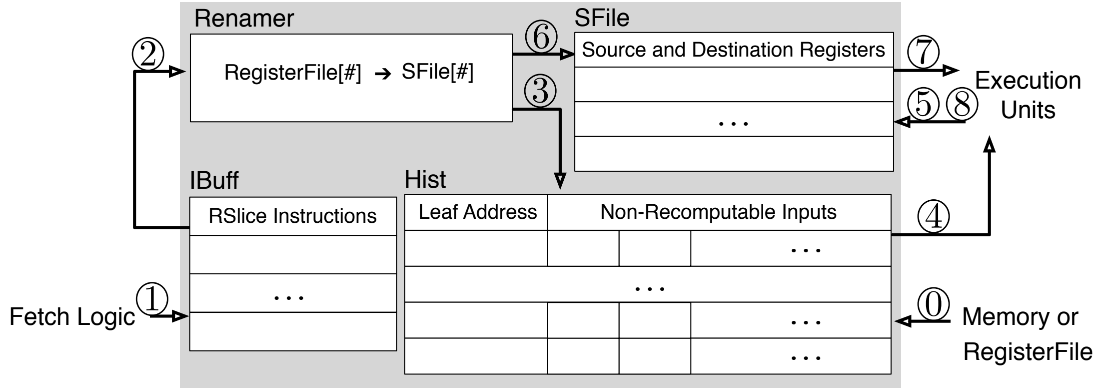

Figure 3.2: Amnesic Microarchitecture & Scheduler.

in *RSlice(v)*. `REC` practically checkpoints the input operands to a dedicated buffer (see Sections 3.3.2 and 3.3.3).

Unless the compiler can prove that all input operands of *RSlice(v)*'s leaves correspond to constants or live register values at the anticipated time of recomputation, `REC` instructions are necessary. Finally, how the compiler orders the leaves in *RSlice(v)* code is not critical, as leaf instructions cannot depend on each other.

3.3.2 Amnesic Microarchitecture

Amnesic execution should meet two conditions for safe and effective recomputation:

Condition-I: Prevent corruption of the architectural state during recomputation (see Section 3.2.3).

Condition-II: Have (non-recomputable) input operand values of *RSlice* leaves available at the anticipated time of recomputation (see Section 3.2.2).

Figure 3.2 captures microarchitectural support to meet **Condition-I** and **Condition-II** in orchestrating amnesic execution. Recall that only one *RSlice* can be active, i.e., traversed for recomputation, at a time[4] .

Scratch-File (SFile): To satisfy **Condition-I**, the amnesic microarchitecture deploys the dedicated buffer SFile. During recomputation, as program control traverses an

[4] Offloading recomputation to spare or idle cores, or using helper threads may improve energy efficiency further by enabling concurrent recomputation. However, the basic proof-of-concept implementation assumes strictly sequential execution semantics.

RSlice, the data flows through the SFile, leaving the (physical) registerfile intact. Recomputing instructions from an *RSlice* do not perform any memory access, and communicate over SFile only.

Renamer: During traversal of each *RSlice*, a dedicated Renamer maps register references per recomputing instruction to SFile entries. Semantically, the amnesic renamer closely mimics the rename logic of classic out-of-order machines. In this context, SFile becomes not any different than the physical registerfile and follows similar rules for space (de)allocation.

History Table (Hist): For each *RSlice* where the leaf input operands correspond to constants or live values from the (physical) registerfile, **Condition-II** is automatically satisfied. Only for non-recomputable leaf input operands, dedicated storage is required to satisfy **Condition-II**. The amnesic microarchitecture can buffer non-recomputable input operands for each *RSlice* leaf in the dedicated history table Hist. Each entry of Hist keeps the address (`leaf-address`) and non-recomputable input operands of a leaf instruction.

Instruction Buffer (IBuff) can cache recomputing instructions within each *RSlice*, in order to relax amnesic execution's potential pressure on the instruction cache. Each entry of IBuff corresponds to a recomputing instruction.

SFile, Hist, and IBuff all feature an `invalid` field per entry to orchestrate (de)allocation of space as necessary.

3.3.3 Amnesic Scheduler

Runtime Policies

At runtime, the amnesic scheduler decides whether recomputation along each *RSlice(v)* embedded into the binary by the compiler (Section 3.3.1) can improve energy efficiency or not, depending on where in the memory hierarchy v resides. Specifically, each time a `RCMP` instruction is fetched, the scheduler has to decide whether to branch to the entry point of the respective *RSlice(v)*, or whether to perform the load to read v from memory. A control flag, `recompute`, remains set as recomputation – traversal of an *RSlice* – is in progress. `recompute` is reset by default.

To be able to draw a safe decision, the amnesic scheduler needs to track where in the

memory hierarchy v resides. There are different options to track or predict the location of v at runtime. In the proof-of-concept implementation, the amnesic scheduler lets the corresponding load probe on-chip memory (caches), and fires recomputation upon a miss in the first-level cache (*FLC*), or alternatively, upon a miss in the last-level cache (*LLC*) – by using either a first or a last level cache miss as an indicator for an energy-hungry off-chip memory access. In this case, `RCMP` becomes the equivalent to `branch on FLC miss` or, alternatively, `branch on LLC miss`, with the branch target being the entry point of the respective *RSlice*. The amnesic scheduler fires recomputation by setting the `recompute` flag. Otherwise, execution follows the classic trajectory by performing the load.

In this case, recomputation cost includes the cost of probing the on-chip memory hierarchy. *FLC* and *LLC* policies are heuristic-based and may result in false-negatives (lost recomputation opportunity) and false-positives (energy-inefficient recomputation). Better amnesic policies can be devised by using more accurate (miss) predictors [7, 8, 9], which can also help eliminate the probing overhead. We leave further refinement and exploration of such policies to future work – the design space is pretty rich. In Section 3.5, we will also compare *FLC* and *LLC* policies to a runtime-oblivious policy, *Compiler*, which *always* triggers recomputation each time a `RCMP` instruction is fetched.

3.3.4 Putting It All Together

Amnesic activity when `recompute` is reset: No recomputation takes place as long as the `recompute` flag stays reset. During this period, amnesic execution is equivalent to classic execution, if no *RSlice* in the binary features non-recomputable leaf inputs. Otherwise, the amnesic scheduler has to record such non-recomputable input operands into Hist. To this end, the scheduler tracks `REC` instructions (Section 3.3.1). `REC` instructs the scheduler to record all non-recomputable input operands in a Hist entry (⓪ in Figure 3.2), along with `leaf-address`.

Triggering recomputation: For each `RCMP` instruction fetch-ed, the amnesic scheduler first needs to resolve the branching condition: whether recomputation is more energy-efficient than performing the memory access, i.e., whether $E_{ld,v}$ exceeds $E_{rc,v}$. This decision can be drawn following any of the runtime policies from Section 3.3.3, *FLC* or *LLC*. For example, under *LLC*, the amnesic scheduler probes the caches, and fires

recomputation by setting the `recompute` flag upon an LLC miss. Otherwise, the load is performed following the classic execution trajectory.

Amnesic activity when `recompute` is set: `RCMP` branches to the entry point of *RSlice(v)*, and instruction fetch starts from the first leaf. Each leaf instruction first has its destination register renamed (② in Figure 3.2). Each leaf instruction with non-recomputable input operands next probes Hist with `leaf-address` (③) to read its input operands, which directly are fed into the corresponding execution units (④). Leaf instructions with constant or live register input operands do not need to probe Hist. Upon finishing execution, each leaf writes its result to the SFile (⑤).

Non-leaf recomputing instructions which represent intermediate nodes in *RSlice(v)* read their input operands from SFile (⑥) after having their source and destination registers renamed (②). Upon collecting the input operands, recomputing instructions proceed to the execution units (⑦), and write their results back to the SFile once execution completes (⑧). All (non-leaf) recomputing instructions in *RSlice(v)* execute sequentially in this manner until the `RTN` instruction of the slice is fetched. Before return, the recomputed data value v gets copied from SFile into the destination register of the eliminated load (recall that `RCMP` inherits all source and destination parameters of the respective load). The amnesic scheduler then resets `recompute` flag to demarcate the end of recomputation. Execution continues from the instruction following `RCMP` in program order.

IBuff is an optional structure to help reduce the pressure on instruction cache under recomputation. Very much like the instruction cache, fetch logic can fill IBuff with recomputing instructions (①). IBuff in turn feeds the Renamer with recomputing instructions (②).

3.3.5 Storage Complexity

We next analyze the expected storage complexity for each component of the amnesic microarchitecture from Figure 3.2. Recall that the amnesic microarchitecture only processes instructions with register source operands and register destinations, and excludes memory or control flow instructions. Without loss of generality, the following analysis assumes a RISC-style ISA.

SFile: A recomputing instruction typically writes its result to one destination register,

and reads its input operands from two source registers. Accordingly, the maximum possible number of renaming requests per recomputing instruction, $max_{\#\text{rename}}$ becomes

$$max_{\#\text{rename}} = max_{\#\text{src}} + max_{\#\text{dest}} = 3$$

where $max_{\#\text{src}}$ ($max_{\#\text{dest}}$) is the maximum number of source (destination) register operands per recomputing instruction. At any given time, only one *RSlice* can be traversed. Therefore, SFile capacity does not depend on the total *RSlice* count in the binary, but grows with the instruction count per *RSlice*, which can exponentially increase with the tree height h. A tall *RSlice*, however, is very unlikely to find any place in the binary, as it can easily result in excessive recomputation overhead to render recomputation useless. The amnesic compiler captures such diminishing returns and prevents excessive growth of the *RSlice* (see Section 3.3.1): practically, the compiler not only influences *RSlice* topology, but also caps the tree height h to maximize energy savings. Accordingly, we can derive a loose upper-bound for SFile capacity as

$$max_{\#\text{inst per } RSlice} \times max_{\#\text{rename}} = max_{\#\text{inst per } RSlice} \times 3$$

where $max_{\#\text{inst per } RSlice}$ corresponds to the maximum of instruction count per *RSlice* across all *RSlice*s in the binary.

Hist: Hist can keep data for multiple *RSlice*s during execution. For each *RSlice*, Hist can contain as many entries as the *RSlice*'s number of leaves. Thus, a loose upper-bound for the number of entries in Hist becomes

$$\#RSlice \times max_{\#\text{leaf per } RSlice}$$

where $\#RSlice$ is the number of *RSlice*s in the binary; and $max_{\#\text{leaf per } RSlice}$, the maximum of the number of leaves per *RSlice* (which may grow with tree height h). Each Hist entry accommodates at most $max_{\#\text{src}}$ values, to cover all non-recomputable input operands per leaf.

IBuff: The capacity of IBuff grows with the number of instructions per *RSlice*. Hence, a loose upper-bound for IBuff capacity becomes $max_{\#\text{inst per } RSlice}$.

3.3.6 Technicalities

The proof-of-concept implementation represents a basic design, which neglects various optimization opportunities such as instruction reuse among recomputing slices, or hardware resource sharing with the underlying microarchitecture.

During traversal of an *RSlice*, latency per recomputing instruction remains very

similar to its classic counterpart, as the amnesic microarchitecture follows the pipelining semantics of the underlying microarchitecture (just with an alternative instruction and operand supply of similar latency).

The storage complexity of amnesic structures from Figure 3.2 tends to be low (Section 3.3.5). Only the unlikely capacity overflow of Hist can impair recomputation, and only for *RSlice*s with non-recomputable leaf input operands. The amnesic scheduler can track these cases by failed `REC` instructions (Section 3.3.1) and enforce the corresponding `RCMP` to skip recomputation (i.e., to perform the load). To this end, the amnesic scheduler has to uniquely identify the matching `RCMP`. This can be achieved by assigning a unique ID, `RSlice-ID`, to each *RSlice* in the binary, and providing it as an operand to both `REC` and `RCMP`.

In processing recomputing instructions, the amnesic microarchitecture has to differentiate between leaves and intermediate nodes, since different structures supply the input source operands to each: The inputs of leaves can come from the registerfile (a live value) or Hist (an overwritten value). The inputs of intermediate nodes come from SFile. The compiler annotates leaves and accesses to Hist to distinguish between these cases. Specifically, the compiler changes source register identifiers of leaf instructions reading their operands from Hist to an invalid number. Leaf instructions with valid source register identifiers directly access the registerfile. Non-leaf recomputing instructions follow the paths ② and ⑥ in Figure 3.2.

Recall that there is another potential class of leaves with non-recomputable input operands: read-only values to be loaded from memory, such as program inputs. In principle, replacing the load to read v from memory with *RSlice(v)* which features possibly more than one such load at the leaves does not make sense. Hist is designated to record overwritten register input operands, but Hist can also keep such read-only values, and may make recomputation along such *RSlice(v)* energy-efficient.

3.4 Evaluation Setup

Benchmarks: To quantify the energy efficiency potential of amnesic execution, we experiment with 33 sequential or single-threaded benchmarks from SPEC-2006 [10], NAS [11], PARSEC [12] and Rodinia [13] suites, which span various application domains

Suite	Benchmarks	Inputs
SPEC	mcf, perlbench, gobmk, calculix GemsFDTD, libquantum, soplex, lbm omnetpp, sphinx3 (sx)	test
NAS	is	A
	cg	W
	ft, mg	S
PARSEC	canneal (ca), facesim (fs), ferret (fe) raytrace (rt), blackscholes, x264 dedup, freqmine, fluidanimate streamcluster, swaptions, bodytrack	simsmall
Rodinia	backpropagation (bp)	65536
	bfs	graph1MW_6.txt
	kmeans	kdd_cup
	nw	2048 10 1
	particlefilter	-x 128 -y 128 -z 10 -np 10000
	srad (sr)	100 0.5 502 458 1
	hotspot	512 512 2 1

Table 3.1: Benchmarks deployed to quantify the potential of amnesic execution.

and memory access characteristics, as listed in Table 3.1.

Binary generation: We implement the greedy compiler pass detailed in Section 3.3.1 as a (*binary generator*) Pin [14] tool. The EPI estimates (see Section 3.3.1) come from measured data from [15]. Although these estimates are for a parallel processor (Intel's Xeon Phi), the simulated microarchitecture is very similar to its per core configuration (Table 3.2). We also fine-tune these estimates by extracting EPI values for different instruction categories from McPAT [16] integrated with the Sniper-6.1 [17] microarchitectural simulator. We derive $\Pr_{Li}$ (see Section 3.3.1), the probability of having a load serviced by level Li in the memory hierarchy, using hit and miss statistics for Li from Sniper. We also implement a *runtime profiler* in Pin, which collects dependency information for binary generation. Using the dependency information (from the Pin-based runtime profiler) and EPI estimates, the (binary generator) Pin tool identifies *RSlice*s that can improve energy efficiency, and instruments them for inclusion into the binary.

Technology node:		22nm	
Operating frequency:		1.09 GHz	
L1-I (LRU):	32KB, 4-way	0.88nJ	3.66ns
L1-D (LRU, WB):	32KB, 8-way	0.88nJ	3.66ns
L2 (LRU, WB):	512KB, 8-way	7.72nJ	24.77ns
Main Memory	Read: 52.14nJ	Write: 62.14nJ	100ns

Table 3.2: Simulated architecture to quantify the potential of amnesic execution.

Recomputation at runtime: We implement the amnesic microarchitecture from Figure 3.2 in Sniper, and run the annotated binaries on it. Sniper facilitates seamless integration with Pin. Runtime energy and performance statistics come from Sniper (+ McPAT) simulations. Table 3.2 gives EPI and (round-trip) access latency for each level in the simulated memory hierarchy. We conservatively model EPI and access latency for Hist after L1-D; for SFile, after the physical registerfile; and for IBuff, after L1-I. Accordingly, we model `RCMP`'s overhead after a conditional branch; `REC`'s, after a store to L1-D; `RET`'s, after a jump.

3.5 Evaluation

3.5.1 Impact on Energy Efficiency

Figure 3.3 captures the impact of amnesic execution on energy-delay product, EDP [18], as a proxy for energy efficiency. The y-axis is normalized to the EDP under classic execution. Out of 33 benchmarks we deployed, only 11 have the potential to provide more than 10% EDP gain. In the following, we will focus on these benchmarks. The rest of the benchmarks did not benefit much from recomputation (only 4 provided more than 5% EDP gain) because they did not have many energy-hungry loads and/or recomputation degraded temporal locality. Recomputation cannot improve energy efficiency of compute-bound applications unless they incorporate a few but very energy-hungry memory references.

In Figure 3.3, we compare representative runtime policies from Section 3.3.3 – *FLC*, *LLC* and *Compiler*, to two oracular policies: *Oracle* and *C(onservative)-Oracle*. *FLC*, *LLC*, *Compiler* and *C-Oracle* select from the very same set of *RSlice*s for recomputation

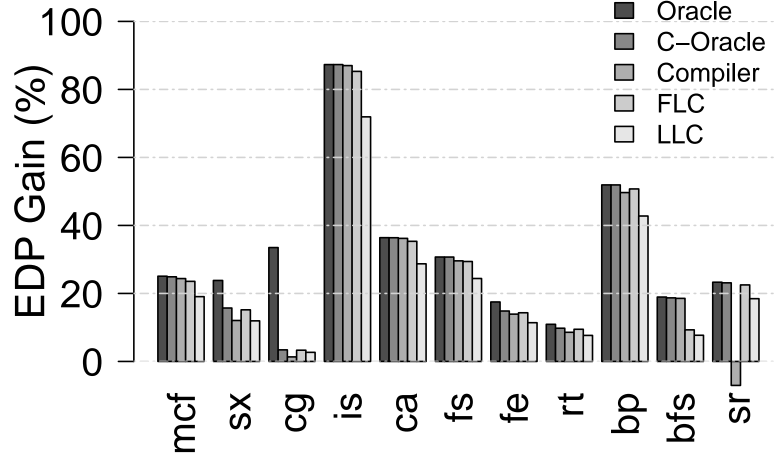

Figure 3.3: EDP gain under amnesic execution.

at runtime – this set is identified by the compiler pass using the probabilistic energy model (see Section 3.3.1). At runtime, *FLC* (*LLC*) fire recomputation along *RSlice(v)* if the respective load of v misses in FLC (LLC). *Compiler*, on the other hand, *always* fires recomputation, for each `RCMP` encountered.

C-Oracle can predict with 100% accuracy where the load of v will be serviced in the memory hierarchy as the amnesic scheduler decides whether to perform the load or whether to fire recomputation along *RSlice(v)*. *C-Oracle* hence bases the runtime decision on this 100% accurate prediction. *Oracle*, too, can predict at runtime with 100% accuracy where a load would be serviced. The key difference of *Oracle* from *C-Oracle* comes from a different (i.e., optimal) set of *RSlice*s baked in the binary, than the compiler's probabilistic energy model based set (which applies to the rest of the policies). The EDP difference between *Oracle* and *C-Oracle* therefore illustrates how accurate compiler's probabilistic energy model is. The smaller the EDP difference, the more accurate is the probabilistic energy model in characterizing an application's loads. In other words, *C-Oracle* demonstrates the maximum possible EDP gain with the given probabilistic energy model of the loads.

We fine-tune the probabilistic energy model of the amnesic compiler pass using dynamic execution traces (see Section 3.3.1). Notice that the EDP gain under *Compiler* evolves with the accuracy of this probabilistic energy model, but such fine-tuning may not always be possible. The more accurate the energy model, the more accurate becomes

amnesic compiler's prediction of where the load reading v will be serviced at runtime. And the more accurate this prediction, the more energy efficiency can the *Compiler* policy harvest, under which each `RCMP` always triggers recomputation. The EDP gains under *Compiler* therefore reflect best-case estimates.

Recall that the set of *RSlice*s recomputed by each policy is different: *Compiler* recomputes along each *RSlice* embedded in the binary, which form the set S. *C-Oracle* picks the optimal subset from S ($S_{C-Oracle}$) for recomputation, i.e., only recomputes *RSlice(v)* if recomputation is exactly more energy-efficient than performing the load of v. *FLC* (*LLC*), on the other hand, picks the subset of S, S_{FLC} (S_{LLC}), which only includes *RSlice(v)*s where the respective load to read v misses in L1 (L2). Subject to the accuracy of the probabilistic energy model and such runtime decisions, the set of *RSlice*s recomputed by *Oracle* may be very different: *Oracle*'s decisions are based on actual (not probabilistic or predicted) energy costs.

Overall, with the exception of sx and cg (and fe, rt to a lower extent), we observe that *C-Oracle* closely tracks *Oracle*, rendering the probabilistic energy model accurate. Except sr, the best-case *Compiler* closely tracks *C-Oracle*. On the other hand, the difference between the best-case *Compiler* and *FLC* is barely visible, with the exception of sx, bfs and sr. *LLC* is consistently worse than *FLC*. The main delimiter for *LLC* is the overhead of probing the last-level cache (L2) to detect a miss which is much larger than the overhead of probing the first-level cache (L1) to detect a miss under *FLC*.

EDP(**Compiler**) $<$ **EDP**(**FLC**): In principle, as the amnesic compiler can only probabilistically take into account where a load might get serviced at runtime, by firing recomputation along *RSlice(v)* for each `RCMP` encountered, the *Compiler* policy can easily trigger unnecessary recomputations, and hence, hurt energy efficiency – particularly if v resides in L1. *FLC*, on the other hand, prevents recomputation in this case. This is clearly visible for sr, where *Compiler* triggers too many recomputations that do not provide sizable energy gain (due to recomputed data mostly being in L1), but introduce performance overhead (since RSlices recomputed usually take longer than accessing L1). Since the energy gain due to recomputation does not offset the performance degradation, the EDP of sr degrades 7% under *Compiler*. Although the difference is small, *Compiler* yields lower EDP gain than *FLC* in sx, cg, fe, rt and bp.

EDP(**Compiler**) $>$ **EDP**(**FLC**): *Compiler* can provide higher gains than *FLC* (*LLC*)

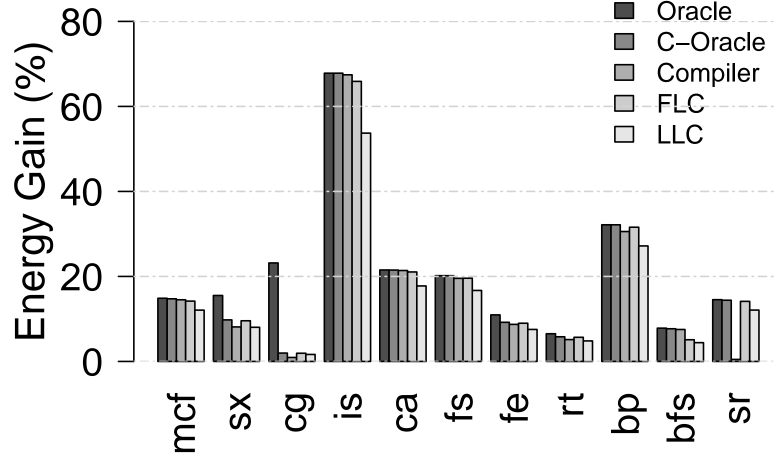

Figure 3.4: Energy gain under amnesic execution.

when they recompute the very same set of *RSlice*s; i.e., S_{FLC} (S_{LLC}) overlaps with S – when none of the vs is present in L1 (L2). This is because *Compiler* does not need to probe the caches, so there is no probing cost. Although the difference is mostly small, this is the tendency in mcf, is, ca, fs, and bfs.

EDP(FLC) vs. EDP(LLC): If v resides in L1, both *FLC* and *LLC* simply skip recomputation. If v resides in L2, only *FLC* fires recomputation. In this case, depending on the instruction mix and count in *RSlice(v)*, recomputation may be less expensive than retrieving v from L2, particularly for short *RSlice(v)*. At the same time, the probing cost is lower for *FLC* than *LLC*. As Section 3.5.4 reveals, the benchmark applications feature predominantly short *RSlice(v)*s, with much less than 50 instructions. Overall, *FLC* renders the higher EDP gain, since recomputation along *RSlice(v)* remains usually cheaper than retrieving v from L2.

Impact on energy & execution time: Due to memory accesses being both energy-hungry and slow, most of the time, the reduction in EDP comes from a reduction in both energy and execution time. Figure 3.4 shows the corresponding reduction in energy consumption; Figure 3.5, in execution time, under amnesic execution, normalized to classic execution. We observe similar trends to EDP for both.

Putting it all together: An amnesic design which always fires recomputation following compiler hints (i.e., *Compiler*, as opposed to following policies like *FLC* or *LLC*) can be very effective as Figure 3.3 reveals, but it is limited by the accuracy of compiler's

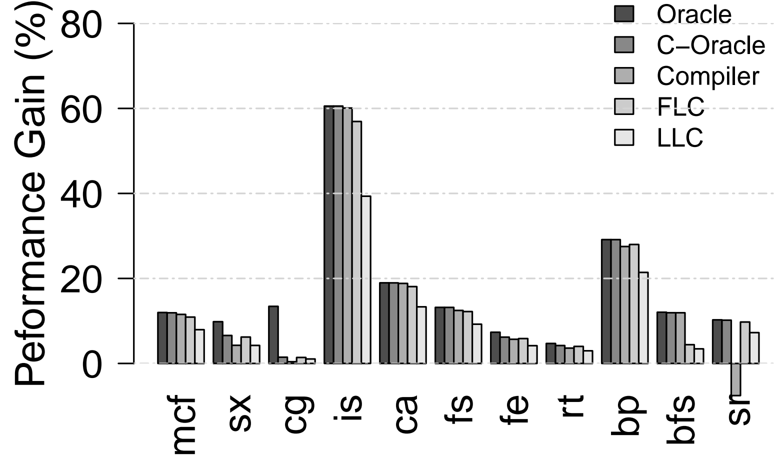

Figure 3.5: % reduction in execution time.

probabilistic energy model. Overall, *Compiler* improves the EDP of all benchmarks, with the exception of sr and mg where EDP is degraded by 7% and 1.37%, respectively. Eight of the benchmarks obtain more than 10% EDP gain under *Compiler*, where the range changes from 12.04% to 87%. *FLC* and *LLC* yield slightly lower EDP gains than *Compiler*, in general. Since they tend do make more conservative decisions on recomputation, they do not experience any EDP degradation. For the aforementioned 8 benchmarks, EDP gain under *FLC* (*LLC*) range from 14.37% to 85.3% (11.39% to 71.92%).

To shed further light on these findings, we will next look into instruction mix (see Section 3.5.2), memory access characteristics (see Section 3.5.3) and *RSlice* characteristics (see Section 3.5.4) under amnesic execution.

3.5.2 Impact on Instruction Count and Mix

Under amnesic execution, the sequence of recomputing instructions in each *RSlice(v)* replaces the respective load to read v from memory. Therefore, we expect an increase in the number of (dynamic) instructions along with a decrease in the number of (dynamic) load instructions under amnesic execution. Table 3.3 shows how the dynamic instruction mix and energy breakdown changes under amnesic execution. For comparison, we also provide the energy breakdown under classic execution. Without loss of generality,

Bench.	% incr. (dyn.) instr. count	% decr. load count	Energy Breakdown (%)						
			Classic			Amnesic			
			Load	Store	Non-mem	Load	Store	Non-mem	Hist Read
mcf	4.47	6.19	91.67	2.12	6.20	75.33	2.88	6.77	0.48
sx	4.55	6.68	70.43	2.70	26.86	58.44	3	28.01	2.42
cg	3.97	2.11	82.43	0.45	17.10	80.03	0.51	17.99	0.51
is	17.97	49.99	84.30	11.19	4.49	9.62	13.17	9.75	3.06e-06
ca	7.38	7.95	85.21	5.16	9.61	62.26	5.20	10.42	0.70
fs	1.83	3.08	53.90	14.37	31.71	32.36	14.78	32.61	0.68
fe	3.55	1.75	58.49	15.50	26	47.81	15.57	27.03	0.84
rt	1.97	6.08	67.87	8.58	23.54	60.67	8.73	24.27	1.16
bp	31.89	55.55	87.71	7.22	5.05	52.68	7.22	7.38	2.13
bfs	1.20	60.93	79.18	1.87	18.94	68.35	2.20	21.92	2.42e-07
sr	20.02	23.33	49.89	9.43	40.66	30.35	14.69	47.11	7.36

Table 3.3: Dynamic instruction mix and energy breakdown under amnesic execution.

we report the amnesic execution outcome for the *Compiler* policy, which incurs the maximum possible number of recomputations.

The first half of the table captures the % increase in the dynamic instruction count along with the % decrease in the dynamic load count under amnesic execution with respect to the classic baseline. In the second half, we report the % energy breakdown under classic and amnesic execution: we differentiate between stores, loads and all other instructions (which form the category *Non-mem*). Under amnesic execution, we also report the share of Hist table reads, which retrieve non-recomputable input operands of *RSlice* leaves.

We observe that amnesic execution reduces the energy consumed by load instructions for all benchmarks, while the energy consumed by *Non-mem* instructions increases due to recomputation along *RSlice*s. is from NAS, among the benchmarks listed in Table 3.3, is the most responsive to amnesic execution: The energy consumption of its loads drops from 84.3% to 9.62%, at the expense of executing $\approx$ 17.97% more instructions due to recomputation. In return, the number of dynamic loads reduces by 49.99% under amnesic execution.

3.5.3 Memory Access Characteristics

The effectiveness of amnesic execution is constrained by, for each target data value v, (i) where in the memory hierarchy v resides; (ii) the cost of recomputation along *RSlice(v)*.

Bench.	Compiler (hit %)			FLC (hit %)			LLC (hit %)		
	L1-hit	L2-hit	Mem-hit	L1-hit	L2-hit	Mem-hit	L1-hit	L2-hit	Mem-hit
mcf	12.02	11.01	76.97	10.73	11.16	78.09	10.73	11.16	78.09
sx	85.33	0.85	13.80	85.08	0.86	14.04	85.09	0.85	14.05
cg	87.49	0.17	12.33	87.49	0.17	12.33	87.49	0.17	12.33
is	49.64	19.25	31.10	49.64	19.25	31.10	49.64	19.25	31.10
ca	27.85	7.50	64.63	27.84	7.51	64.64	27.84	7.51	64.64
fs	56.47	1.92	41.59	56.46	1.92	41.60	56.46	1.92	41.61
fe	63.26	10.06	26.67	63.22	10.07	26.70	63.22	10.05	26.71
rt	92.95	0.75	6.28	92.21	0.83	6.94	92.85	0.06	7.07
bp	72.49	4.11e-3	27.49	72.49	4.11e-3	27.49	72.49	4.11e-3	27.49
bfs	98.43	1.15e-3	1.56	98.43	1.15e-3	1.56	98.43	1.15e-3	1.56
sr	93.70	0.03	6.26	93.70	0.03	6.26	93.70	0.03	6.26

Table 3.4: Memory access profile of load instructions under classic execution, which are swapped for recomputation under *Compiler*, *FLC*, and *LLC* policies, respectively.

(i) sets the budget for recomputation, and recomputation is only effective if (ii) remains below this budget. The lower the level in the memory hierarchy where v resides, the higher becomes the budget for recomputation along *RSlice(v)*. Amnesic execution is more likely to provide higher energy efficiency, if the target v resides in lower levels of the memory hierarchy.

Table 3.4 shows the memory access profile of load instructions under classic execution, which are swapped for recomputation under *Compiler*, *FLC*, and *LLC* policies, respectively. We report the percentage of such load instructions serviced by each level in the simulated memory hierarchy (Table 3.2). Recall that the set of *RSlice*s recomputed by each policy is different (see Section 3.5.1), therefore, so is the set of loads swapped for recomputation.

Memory access characteristics help us reason about why some benchmarks benefit more from recomputation, considering different policies. For example, bfs exhibits higher EDP gain for the *Compiler* policy, but relatively lower EDP gain for *FLC* and *LLC* policies (Figure 3.3). As Table 3.4 reveals, bfs's swapped loads are almost entirely serviced by L1. Since bfs's swapped loads barely miss in L1, *FLC* and *LLC* policies fire recomputation less often. *Compiler*, on the other hand, triggers recomputation regardless of where the target data resides in the memory hierarchy. bfs's energy efficiency

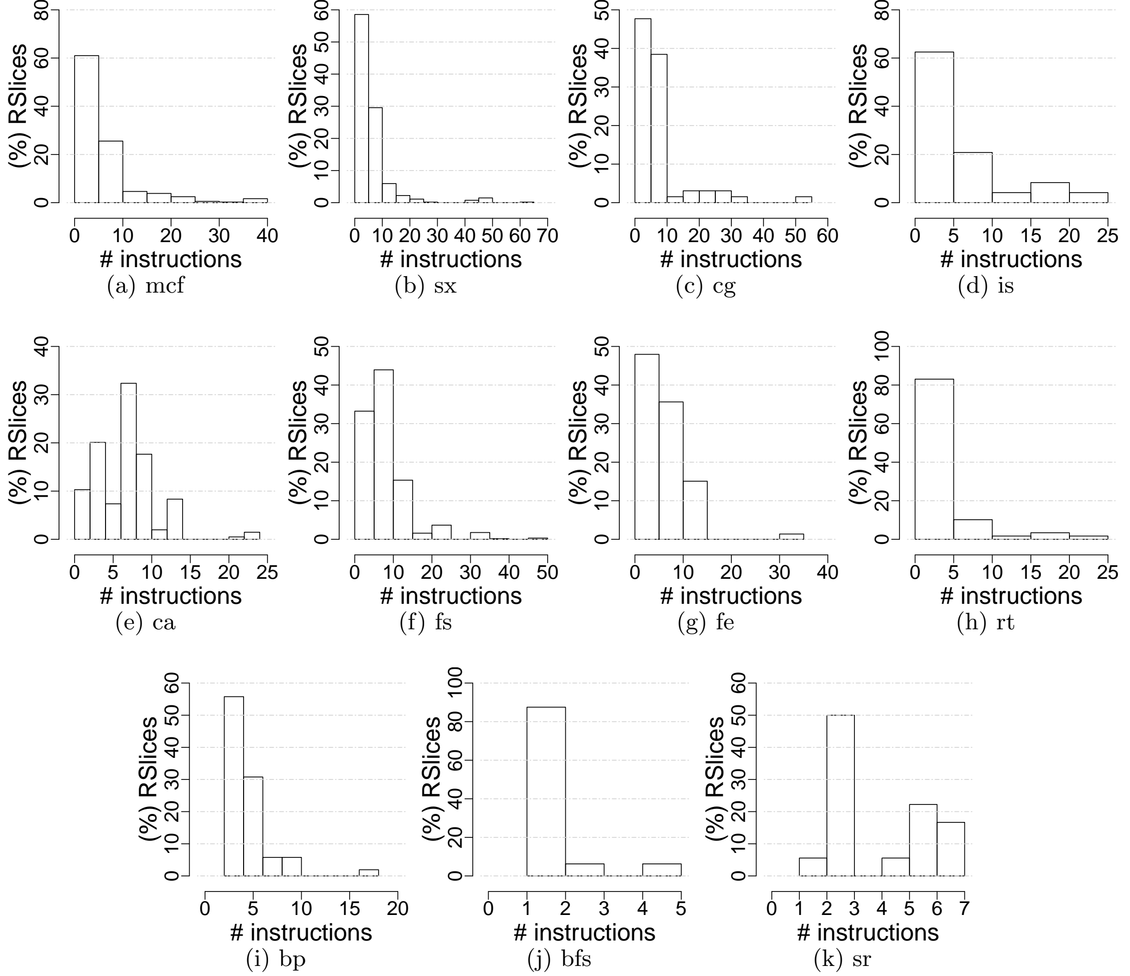

Figure 3.6: Histograms of instruction count per recomputed *RSlice* under *Compiler* policy.

gain under *Compiler* comes from the relatively short, hence cheap *RSlice(v)*s (see Section 3.5.4), even though the target v could be found in L1 most of the time. In this case, *Compiler* comes very to close *Oracle*.

Quite the opposite trend applies for sr, the benchmark where *Compiler* falls noticeably behind *Oracle* and even degrades the EDP. As Table 3.4 reveals, similar to bfs, most (93.7%) of sr's swapped loads are serviced by L1. As it was the case for bfs, the

target v could be found in L1 most of the time, but *Compiler* always triggers recomputation along *RSlice(v)* instead. As the respective *RSlice(v)*s of sr are not as short, hence cheap, as the ones of bfs (see Section 3.5.4), such excess recomputations cause *Compiler* to render a 7% degradation of EDP.

3.5.4 *RSlice* Characteristics

The number of instructions in an *RSlice* (i.e., *RSlice* length) is a fundamental determinant of the cost of recomputation. As *RSlice* length increases, recomputation incurs a higher cost due to the (re)execution of a larger number of instructions. Recomputation, i.e., traversal of an *RSlice(v)* under amnesic execution, provides higher energy efficiency benefits if the target data value v resides in lower levels of the memory hierarchy, and, at the same time, if the respective *RSlice(v)* is relatively short.

Figure 3.6 shows histograms of instruction count per (recomputed) *RSlice* under *Compiler* policy. Recall that *Compiler* always triggers recomputation, independent of where v resides in the memory hierarchy. Therefore, Figure 3.6 covers the profile for the *entire* set of *RSlice*s (as identified by the amnesic compiler; Section 3.3.1). Overall, we observe that 78.32% of the *RSlice*s have a length less than 10 instructions, across the board. Only 0.09% of the *RSlice*s contain more than 50 instructions. According to the storage complexity analysis from Section 3.3.5, this implies a small footprint for SFile and IBuff (Figure 3.2), which grow with *RSlice* length.

For example, for the is benchmark from NAS, more than 30% of the loads swapped for recomputation have their data residing in the main memory (Table 3.4). At the same time, as Figure 3.6d reveals, the application features mostly short *RSlice*s. As a result, amnesic execution results in very high EDP gain (87% according to Figure 3.3). Although bfs features much shorter *RSlice*s than is (Figure 3.6j), its EDP gain remains significantly lower (18.54% according to Figure 3.3), because 98.43% of its loads swapped for recomputation have their data residing in L1 (Table 3.4).

Hist from Figure 3.2 only serves buffering non-recomputable (nc) leaf input operands of *RSlice*s. Figure 3.7 shows the percentage share of *RSlice*s featuring non-recomputable leaf input operands for all applications. With the exception of is and bfs, such *RSlice*s represent the vast majority, rendering Hist a critical structure. According to our analysis, across all benchmarks, Hist has to record the non-recomputable inputs of at most

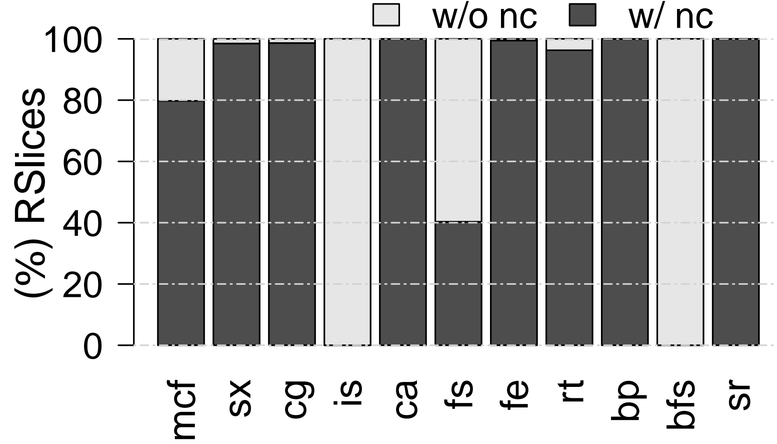

Figure 3.7: % of *RSlice*s with **n**on-re**c**omputable leaf inputs.

565 of such *RSlice*s at a time (i.e., for *fs*), where the average number of leaves is 1. A Hist design of no more than 600 entries can accommodate such demand (see Section 3.3.5).

In the evaluation, we sized the microarchitectural components of Figure 3.2 conservatively for the worst-case, to be able to capture the impact of recomputation without any bias. However, as Figure 3.6 reveals, less than 50 entries for SFile or IBuff can cover most of the *RSlice*s. In this case, recomputation along excessively long *RSlice*s will not be possible, but long *RSlice*s are unlikely to deliver noticeable gains due to the higher (recomputation) cost incurred. Hence, we expect the gains from Figure 3.3 mostly hold under practical sizing considerations.

3.5.5 Break-even Point

The basic idea behind amnesic execution is to swap energy-hungry load instructions with a sequence of non-memory (*Non-mem*) instructions to generate the respective data values. Each such sequence forms an *RSlice*. The non-memory instructions in an *RSlice* are mostly arithmetic/logic, as *RSlice*s do not feature memory or control flow instructions by construction (see Section 3.3.1). The effectiveness of amnesic execution hence comes from such non-memory instructions being significantly less energy-hungry than load instructions, in today's machines at least.

The energy efficiency gain under amnesic execution tightly depends on the relative

Benchmark	$\mathbf{R_{breakeven}}$ (normalized)
mcf	66.74
sx	53
cg	22.89
is	73.74
ca	30.71
fs	32.35
fe	13.7
rt	45.63
bp	83.25
bfs	3.89
sr	36.74

Table 3.5: Break-even point (for *C-Oracle*).

energy cost of non-memory instructions with respect to loads, i.e.,

$$R = EPI_{Non-mem}/EPI_{ld}$$

where $EPI_{Non-mem}$ captures the average EPI of a non-memory (i.e., arithmetic/logic) instruction; EPI_{ld}, of a load. R is a strong function of the underlying (micro)architecture and technology. The default value of R we used throughout the evaluation is

$$\begin{aligned} R_{default} &= EPI_{Non-mem,default}/EPI_{ld,default} \\ &= 0.45nJ/52.14nJ \approx 0.0086 \end{aligned}$$

which comes from the measured EPI estimates from [15] (see Section 3.4). We next extract the value of R which would render amnesic execution useless, i.e., which would result in the same EDP under amnesic and classic execution. In other words, we analyze by how much the relative energy cost of non-memory instructions should increase (with respect to loads) to reach the break-even point for amnesic execution.

As the relative energy cost, R, increases, amnesic execution becomes less and less beneficial, and past the value of R at the break-even point, $R_{breakeven}$, as expensive as classic execution. Table 3.5 lists $R_{breakeven}$, normalized to $R_{default}$, for all of the benchmark applications. Each benchmark application reaches the break-even point at a different value of R due to the differences in the instruction mix (and hence, in *RSlices*). For example, for bfs to reach the breakeven point, R (the relative cost of a non-memory instruction with respect to a load) should increase by 3.89× over its default, $R_{default}$. $R_{breakeven}/R_{default}$ takes much higher values for the rest of the benchmarks.

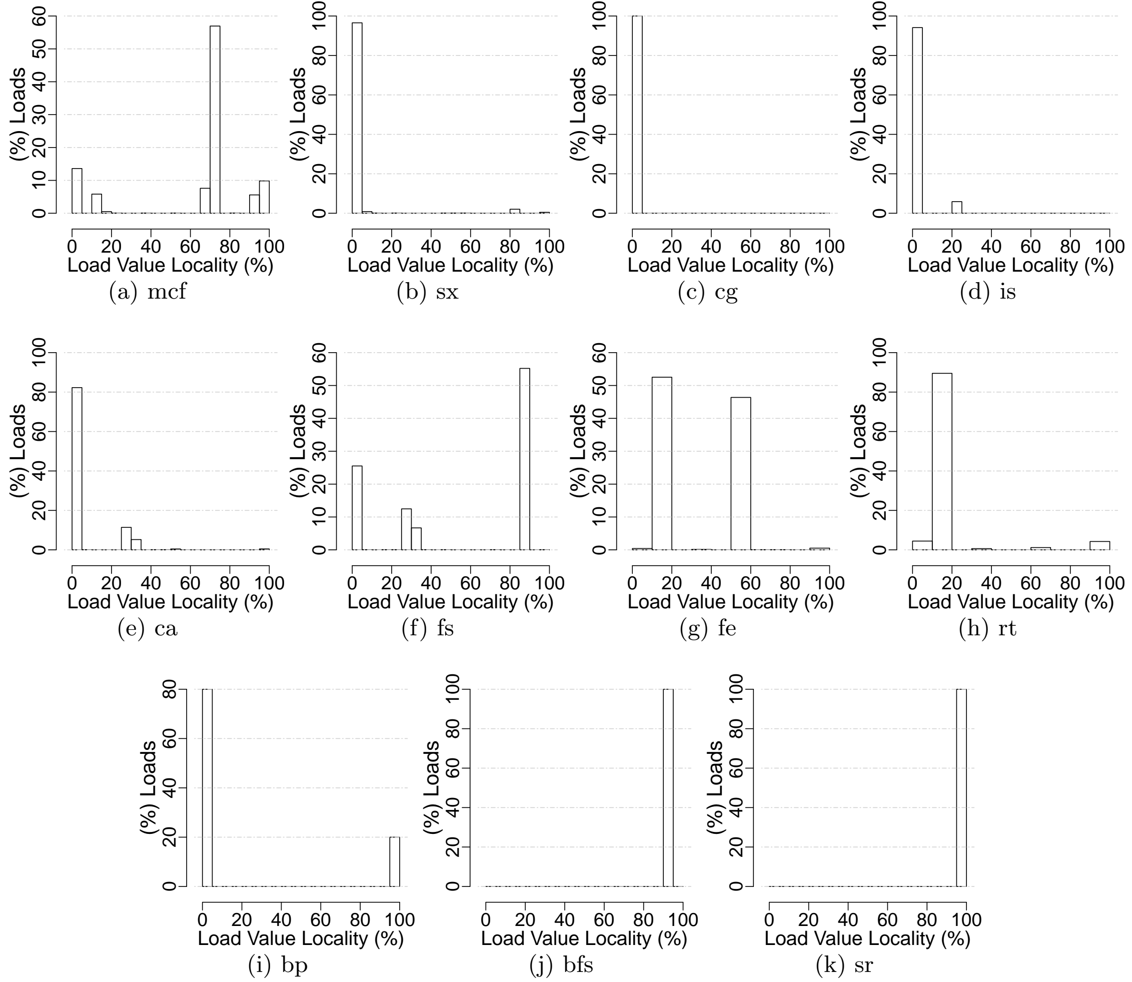

Figure 3.8: % value locality of loads (under classic execution), which are swapped for recomputation by the *Compiler* policy.

In conclusion, unless R increases over $R_{default}$ by the coefficients provided in Table 3.5, amnesic execution is likely to stay more energy-efficient than its classic counterpart. Considering current technology projections [4], such increases are unlikely.

3.5.6 Data Locality Analysis

Figure 3.8 shows the % value locality of load instructions, which are swapped for recomputation under the *Compiler* policy. In other words, these are the loads which get replaced by *RSlice*s. Without loss of generality, we stick to the *Compiler* policy in order to cover the entire set of swapped loads – recall that *FLC* and *LLC* policies only selectively swap loads for recomputation, while *Compiler* always enforces the swap.

We observe that, except bfs and sr, all of the benchmarks exhibit relatively low value locality for the swapped loads – the percentage of the swapped loads that have higher than 95% value locality remains less than 28% across the board. For bfs and sr, all of the swapped loads exhibit around 90% (Figure 3.8j) and 99% (Figure 3.8k) value locality, respectively. For cg, value locality is practically 0% (Figure 3.8c).

This analysis indicates that amnesic execution is mostly *orthogonal* to alternative approaches such as load value prediction [19, 20] or memoization which exploit value locality to mitigate communication overhead. Memoization represents the dual of recomputation: the idea is replacing frequent and expensive computation with table look-ups for pre-computed data. In this manner, memoization can mitigate the communication overhead, since table look-ups are much cheaper than long-distance data retrieval. However, memoization is only effective if the data values generated by the respective computations exhibit significant value locality – in our context, these computations correspond to recomputation along *RSlice(v)*s to generate the data values v, and we capture in Figure 3.8 the locality of such v by the value locality of the respective loads to read v from memory, without loss of generality.

3.6 Related Work

Algorithmic level optimizations to reduce communication is extensively explored in the literature, especially in scientific computing domain [21, 22, 23].

Due to the limited number of registers, and the increasing volume of data to process, the compiler often confronts the NP-complete register allocation problem [24]. A classic compiler optimization during register allocation, **Rematerialization** [25] can eliminate the spilling-induced store and replace spilling-induced (consumer) loads by a sequence of instructions to recompute the value (that would be spilled otherwise), provided that

the input values needed for recomputation are ready at the time of recomputation, and recomputation is more cost-effective where the cost is defined in terms of latency. This pass inherently ensures that the recomputing instructions do not overwrite register values in use.

Kandemir et al. proposed recomputation to reduce off-chip memory area in embedded processors [26]. Koc et al. investigated how recomputation of data residing in memory banks in low-power states can reduce the energy consumption [27], and devised compiler optimizations for scratchpads [28]. These compiler strategies are limited to array variables. Amnesic execution is not necessarily confined to static compiler analysis or specific data structures. At the same time, as opposed to amnesic execution, these studies fail short of exploring opportunities for hardware-software codesign.

DataScalar [29] trades computation for communication by replicating the most frequently accessed pages in each processor's local memory in a distributed system. As opposed to DataScalar, amnesic execution leverages recomputation at a much finer microarchitectural granularity.

Near memory processing (NMP) [30, 31, 32, 33, 34, 35, 36] can bridge the gap between logic and memory efficiencies by embedding computation capability in main memory. Similar to amnesic execution, NMP can minimize energy-hungry data transfers. Amnesic execution and NMP are orthogonal, and NMP can benefit from amnesic execution to boost energy efficiency, or to reduce the memory footprint.

Memoization [37, 38], the dual of recomputation, replaces (mainly frequent and expensive) computation with table look-ups for pre-computed data. Similar to NMP and amnesic execution, memoization can mitigate the communication overhead, since table look-ups are much cheaper than long-distance data retrieval. Memoization is only effective if the respective computations exhibit significant value locality. Therefore, memoization and recomputation can complement each other in boosting energy efficiency.

Idempotent Processors [39] execute programs as a sequence of compiler-constructed idempotent (i.e., re-executable without any side effects) code regions. *RSlice*s aren't required to be strictly idempotent, but idempotent regions can act as *RSlice*s.

Variants of **Speculative Precomputation** [40, 41, 42, 43, 44, 45] rely on speculative helper threads which run along main threads of execution to enhance performance

(by e.g., masking long latency loads from main memory). Prefetching by helper threads can result in notable performance boost, however, helper threads still perform costly (main) memory accesses. The redundancy in execution incurs a power overhead on top.

Chapter 4

Recomputation Taxonomy

4.1 Introduction

In its simplest form, recomputation entails brute-force *recalculation* on demand, to prevent expensive data transfers. Due to the increasing power and latency gap between computation and data orchestration, recomputation can enhance energy efficiency even in this simplest form. Expanding recomputation to value *prediction* [46, 19] or *approximation* [47, 20] – as long as the underlying potential loss in computation accuracy remains at acceptable levels – can help reduce the input data retrieval overhead of *RSlice*'s leaves under *recalculation*.

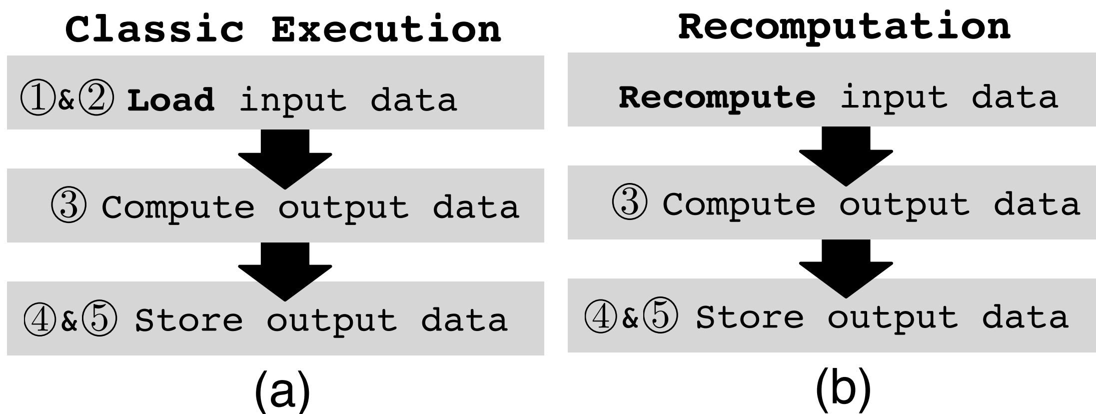

Figure 4.1: Classic execution vs. Recomputation.

Figure 4.1(a) shows the classic trajectory at each step of a typical execution. Black arrows point to the direction of data flow. As depicted in Figure 4.1(b), *recomputation* swaps load instructions for the reproduction of the respective input operands (which would otherwise be loaded from memory) for the subsequent computation. ① incurs the time and power overhead of the memory (hierarchy) access to perform the load; ②, of the subsequent communication of inputs to compute resources. *Recomputation transforms the overhead of ① & ② to the overhead of the recomputation of the respective data values, i.e., of ③.* Therefore, recomputation can only improve energy efficiency if the cost of data reproduction remains less than the cost of ① & ②. In other words, the cost of ① & ② sets the budget for recomputation.

Recomputation can also reduce the pressure on memory capacity and communication bandwidth. A recomputing processor can accommodate more compute resources (in the form of general-purpose cores or specialized accelerators) to occupy the area once allocated to memory (hierarchy). At the same time, under recomputation the workload becomes more compute-intensive to make a better use of classic processors optimized for compute performance, as opposed to energy efficiency. In this chapter, we introduce a taxonomy for recomputation and provide a quantitative comparison.

4.2 Recomputation Taxonomy

The energy cost of the load from Figure 4.1(a) determines the energy budget for recomputation. Unless the energy cost of reproducing data remains less than the energy cost of the respective load, recomputation cannot improve energy efficiency. Whether recomputation can improve energy efficiency or not tightly depends on where the data reside in the memory hierarchy – it is the location of the data in the memory hierarchy which determines the energy cost of the load. On the other hand, recomputation also incurs an energy cost due to the introduction of *recomputing* instructions that produce the data which would otherwise be loaded.

The taxonomy of recomputation techniques spans three dimensions. Recomputation can reproduce the data (which otherwise would be loaded from the memory hierarchy) by: **(I)** brute-force **recalculation** [48]; **(II) prediction** [46, 19]; or **(III) approximation** [47, 20], respectively:

(I) Under brute-force **recalculation**, the recomputation effort goes to the *reproduction of data values* – which would otherwise be loaded from the memory (hierarchy) – by re-executing the recomputing instructions.

(II) Under **prediction**, the recomputation effort goes to the *estimation of data values* by exploiting *value locality* – the likelihood of the recurrence of data values [19] within the course of execution.

(III) Under **approximation**, the recomputation effort goes to the actual *calculation of data values* – as it is the case for brute-force recalculation, however, *at reduced accuracy*. In this case, the compute resources perform recomputation at reduced-accuracy, by e.g., omitting a subset of recomputing instructions which have negligible impact on the accuracy of data values.

Prediction or **approximation** may degrade accuracy of the end results at various degrees, which is not the case for brute-force **recalculation**. In this study, we focus on **recalculation** and **prediction** (without accuracy loss), and leave **approximation** based recomputation to future work.

4.2.1 Recalculation Based Recomputation

Recalculation can be implemented in various ways. The following analysis relies on a compiler-assisted proof-of-concept implementation, following Chapter 3. During code generation, the compiler replaces each energy-hungry load instruction with a sequence of (arithmetic/logic) recomputing instructions which can (re)produce the respective data values. To this end, the compiler recursively traces data dependencies.

In the proof-of-concept implementation, the compiler is in charge of making sure that all input operands of producer instructions within an *RSlice* are available at the anticipated time of **recalculation**. Unless the compiler guarantees this constraint, an *RSlice* cannot replace its respective load in the binary. Further, the compiler swaps a load with its respective *RSlice* only if **recalculation** of the corresponding data value along the *RSlice* is more energy efficient than performing the load.

4.2.2 Prediction Based Recomputation

Under **prediction**, the recomputation effort goes to the estimation of data values, instead of brute-force **recalculation**. Accurate estimation is only possible if data values (which otherwise would be loaded from memory) exhibit high value locality – i.e., a high likelihood of recurrence [19] within the course of execution. For example, if a data value exhibits excellent (100%) locality, just storing the value in a dedicated buffer and retrieving it from there may turn out to be more energy efficient than recalculating it or loading it from memory. Even if the value locality remains less than 100%, such buffered history of values can be used for **prediction**. It has been shown that emerging applications can oftentimes mask prediction incurred inaccuracy due to potential errors in estimation, as implied by imperfect value locality [19].

Value retrieval from the history buffer constitutes the main cost of **prediction**. Under imperfect value locality, a prediction algorithm can help estimate the respective value by using the buffered history of previously observed values. In this case, the cost of executing the prediction algorithm should also be considered. The overall cost of **prediction** should fit into the recomputation budget, which in turn is set by the energy overhead of the respective load. **Prediction** based recomputation can only be beneficial if its energy cost remains less than the energy cost of this load.

4.2.3 Recalculation + Prediction Based Recomputation

Prediction based recomputation (see Section 4.2.2) exploits locality of data values which would otherwise be loaded from memory. With respect to **recalculation** (see Section 4.2.1), **prediction** targets the value to be produced by the (instruction at the) root node of the *RSlice*. Input (data operand) values of *RSlice* nodes may also exhibit significant value locality. Let us assume that such a node n resides at level l, and it is not a leaf. In this case, predicting n's inputs may turn out to be more energy efficient than re-executing producers (of n's inputs) residing at level $l+1$ of the *RSlice*. Hence, combining **recalculation** with **prediction** (i.e., **recalculation + prediction**) can result in pruned *RSlice* to harvest even more energy efficiency. **Prediction** can also serve identifying the inputs of leaves – recall that, if retrieving input data of leaves requires energy hungry memory accesses, recalculation along the *RSlice* cannot be of any

use. Each intermediate node of the *RSlice* subject to **prediction** becomes practically a leaf, as re-execution past such nodes would no longer be necessary.

Recalculation + prediction can prune *RSlice*s, however, even under pure **recalculation** (see Section 4.2.1), *RSlice*s can never grow excessively: the energy cost of the respective load determines the budget for recomputation. The cost of **recalculation** increases with the number of levels, i.e., *height* of the *RSlice*, and the number of nodes residing at each level. The re-execution of each node instruction incurs an energy cost. At most, as many nodes can be re-executed (i.e., can reside in the *RSlice*) as can be fit into the recomputation budget. And **recalculation** can only improve energy efficiency if the cost of re-execution along the *RSlice* remains less than the recomputation budget, which is set by the energy cost of the respective load. In this manner, the energy cost of the load prevents excessive growth of the *RSlice*. Under **recalculation + prediction**, the cost of re-execution along the *RSlice* along with the cost of selective **prediction** constitute the cumulative cost of recomputation.

4.3 Evaluation Setup

We experiment with benchmarks from the SPEC2006 [10], PARSEC [12], NAS [11], and Rodinia [13] suites, which span emerging applications (Table 4.1). In the evaluation, we only analyze the benchmarks which harvest sizable (i.e., greater than 10%) energy efficiency gain under recomputation. The analyzed mix contains both compute- and memory-intensive applications. Our analysis is confined to sequential, i.e., single-threaded execution. We use the Sniper [17] micro-architectural simulator. We use the same microarchitectural configuration and simulation infrastructure presented in Section 3.4 for our evaluations. We profile the native binaries (conforming to classic execution, hence excluding recomputation) of the benchmarks on Sniper: We record (i) value locality of instructions at runtime (to be exploited by **prediction** based recomputation); (ii) cache statistics, i.e., hit and miss rates, at runtime (to derive a probabilistic energy cost model for the compiler pass covered in Section 3.3.1).

Suite	Benchmark	Input	Application
SPEC	429.mcf (mcf)	test	Combinatorial Optimization
	482.sphinx3 (sx)	test	Speech Recognition
NAS	is	A	Integer Sorting
PARSEC	canneal (ca)	simsmall	Simulated Annealing
	facesim (fs)	simsmall	Motion Simulation
	ferret (fe)	simsmall	Content Similarity Search
	raytrace (rt)	simsmall	Real-time Raytracing
Rodinia	backpropagation (bp)	65536	Pattern Recognition
	breath-first search (bfs)	graph1MW_6.txt	Graph Traversal
	srad (sr)	100 0.5 502 458 1	Image Processing

Table 4.1: Benchmarks deployed to quantify the potential of different recomputation techniques.

4.4 Evaluation

We next quantify the energy efficiency under recomputation and analyze the implications for execution semantics.

4.4.1 Impact on Energy and Performance

Figure 4.2 compares the energy consumption under **recalculation**, **prediction**, and **recalculation+prediction** based recomputation. This analysis accounts for the overhead of recomputing producer instructions (along *RSlice*s) under **recalculation** (Section 4.2.1), and history buffer accesses under **prediction** (Section 4.2.2). However, we assume that one history buffer access suffices for value prediction at 100% accuracy (i.e., we omit any potential overhead due to prediction algorithms). For this experiment, we set the value locality threshold to enable prediction to 90%: prediction only applies to instructions, the input operands of which exhibit at least 90% value locality. **Prediction** targets only the values to be re-produced by *root* instructions of *RSlice*s (all instructions along which are re-executed under **recalculation**). Under **recalculation+prediction**, on the other hand, prediction can target any RSlice instruction but the root (Section 4.2.3).

Figure 4.2 reports the energy gain with respect to native execution, which excludes

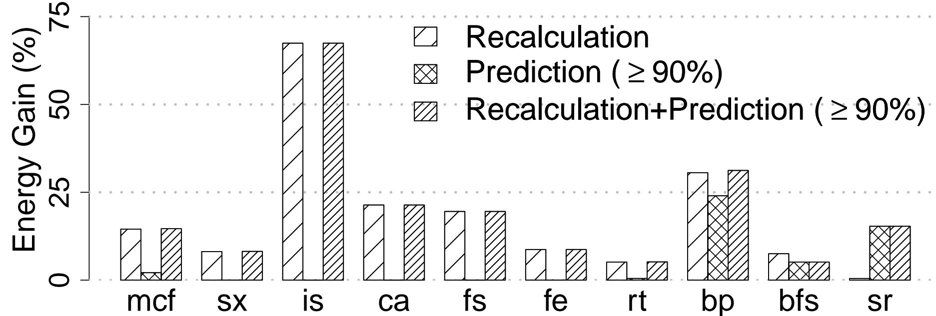

Figure 4.2: Energy gain under recomputation.

recomputation. We observe that except bp, bfs and sr, the energy gain under **prediction** is insignificant. This is because only a small of number of instruction input operands exhibit a higher value locality than 90%. Due to its wider applicability, **recalculation** unlocks higher energy gains, ranging from 5.15% to 67.43%, except sr. The **recalculation** cost for sr remains generally higher than the cost of the respective loads. An interesting observation is that bfs obtains lower energy gain under **prediction** and **recalculation+prediction** when compared to **recalculation** alone. The reason is that the *RSlice*s of bfs are very short, rendering **recalculation** always cheaper than **prediction**. At the same time, our proof-of-concept implementation gives the priority to prediction, if a value exceeds the locality threshold set for prediction (i.e., 90%) under **recalculation+prediction**: in other words, we omit recalculation for all values that exhibit a higher value locality than the threshold (90% in this case), even though recalculation turns out to be less energy hungry. Overall, the energy gain due to **recalculation+prediction** remains limited for the majority of the benchmarks. The reason is twofold: the benchmarks either do not have enough value locality to exploit prediction (e.g. mcf, sx, is, ca, fs, fe, and rt), or recalculation is too costly (e.g. ca, fs).

Figure 4.3 reports the corresponding improvement in performance (i.e., execution time) with respect to native execution. Generally, a similar trend to energy gain applies, except that the performance degrades under **recalculation** for sr, due to recomputed data mostly being in L1 and recalculation introduces performance overhead (since re-execution along *RSlice*s usually takes longer than accessing L1).

Figure 4.4 summarizes the resulting gain in energy efficiency in terms of EDP

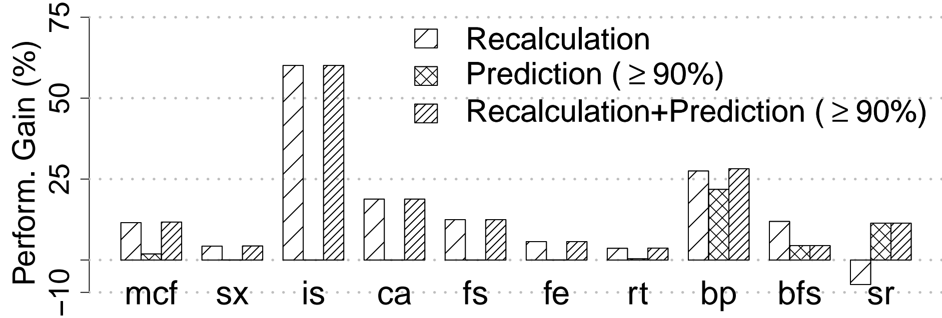

Figure 4.3: Performance gain under recomputation.

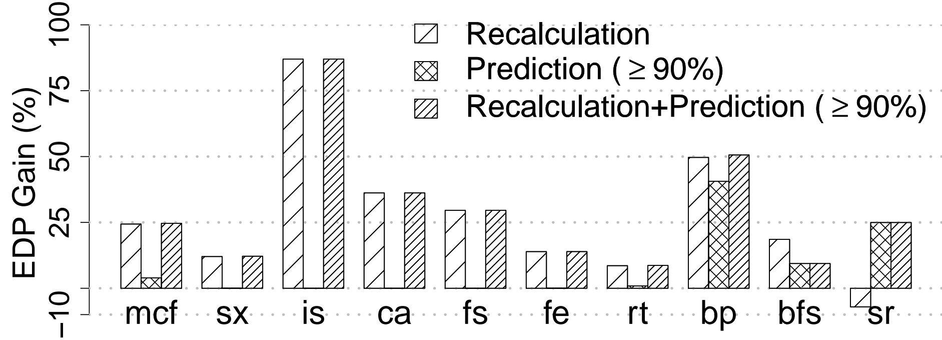

Figure 4.4: EDP gain under recomputation.

(energy delay product [18]), with respect to native execution. Overall, **recalculation+prediction** maximizes the EDP gain, and **recalculation** remains effective as well, except sr (as explained above). **Prediction** is beneficial for bp, bfs, and sr only – recall that even this gain under **prediction** is optimistic as we neglect any algorithmic overhead. Finally, **recalculation+prediction** results in 8.66% to 87% EDP gain across all benchmarks.

We next assess the sensitivity of EDP gain to the value locality threshold for prediction. Figure 4.5 reports the EDP gain under **prediction**; Figure 4.6, under **recalculation+prediction**, as we sweep the threshold between 50% and 100%. Each bar per benchmark represents a different value locality threshold from this range to enable prediction.

Generally, as the threshold increases, the number of instructions exhibiting at least

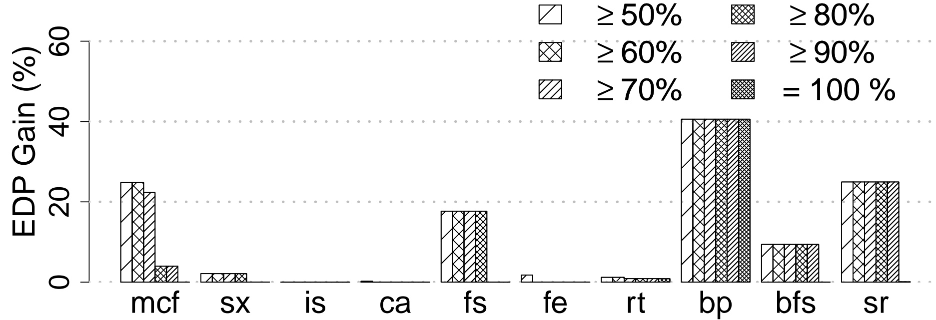

Figure 4.5: EDP gain under **prediction** as a function of value locality threshold for prediction.

that much locality reduces – therefore, a lower number of predictions can be performed, and both the energy and performance gains drop accordingly. Among the benchmarks, bp exhibits the highest value locality, hence, it benefits most from **prediction**. bfs and sr, as well, benefit from **prediction** if the threshold remains lower than 100% – as a very small number of loads swapped for *RSlice*s feature 100% value locality for these benchmarks. On the other hand, fs and mcf harvest sizable EDP gain under **prediction** only if the threshold remains lower than 90% and 80%, respectively. The remaining benchmarks have a very small number of load instructions that exhibit $\geq 50\%$ value locality, so only a negligible EDP gain applies under **prediction** (which already represents an upper limit for actual gains, as we neglect any algorithmic overhead). Therefore, **recalculation+prediction** can generally provide higher EDP gains when compared to **prediction**. As mentioned before, bfs has small *RSlice*s, thus, the associated recalculation cost usually remains lower than than the cost of prediction. Accordingly, bfs shows higher EDP gain for 100% threshold (at which a smaller number of values can be predicted, by definition, when compared to lower values of the threshold) under **recalculation+prediction**. Overall, we observe that our findings from Figure 4.4 generally apply over this wider range of threshold values. We can conclude that *recalculation has wider coverage for recomputation than prediction.* Next, we investigate why this is the case.

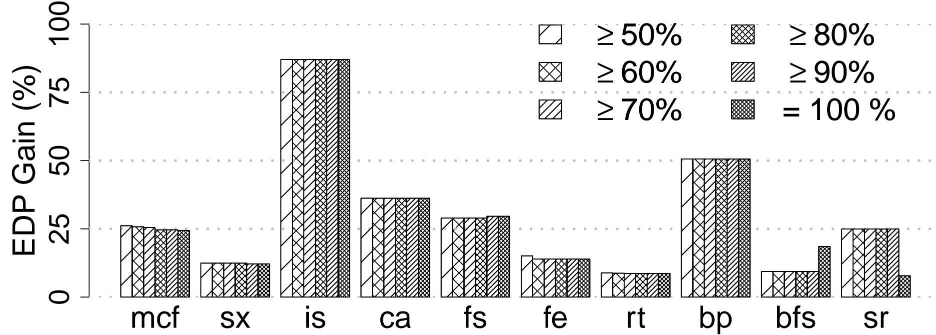

Figure 4.6: EDP gain under **recalculation+prediction** as a function of value locality threshold for prediction.

4.4.2 Impact on Execution Semantics

As explained in Sections 4.2.2 and 4.2.3, in the context of recomputation, prediction serves two purposes:
(i) to predict the values which would otherwise be loaded from memory (and which correspond to the values to be re-produced by *RSlice* roots under pure **recalculation**) under **prediction**;
(ii) to predict the input values of intermediate (non-root) *RSlice* nodes under **recalculation+prediction**.

Prediction can eliminate re-execution along an entire *RSlice* if the values to be re-produced by the *RSlice* root (i.e., the values which would otherwise be loaded from memory) exhibit sufficiently high locality. **Recalculation+prediction**, on the other hand, can prune any intermediate *RSlice* node (along with the attached sub-*RSlice* except the root) exhibiting sufficient (input) value locality to render a smaller *RSlice*, which in turn becomes less energy costly to execute.

For prediction based recomputation to work, the respective instructions should exhibit sufficiently high value locality. Figure 4.7 reports a histogram of % value locality (x-axis) for all instructions residing in *RSlice*s. The y-axis reports the % share of instructions exhibiting a given value of locality on the x-axis. *Root* captures the output value locality of *RSlice* roots; *Non-root*, the input value locality of intermediate (non-root) *RSlice* nodes. Recall that the output value locality of *RSlice* roots corresponds to

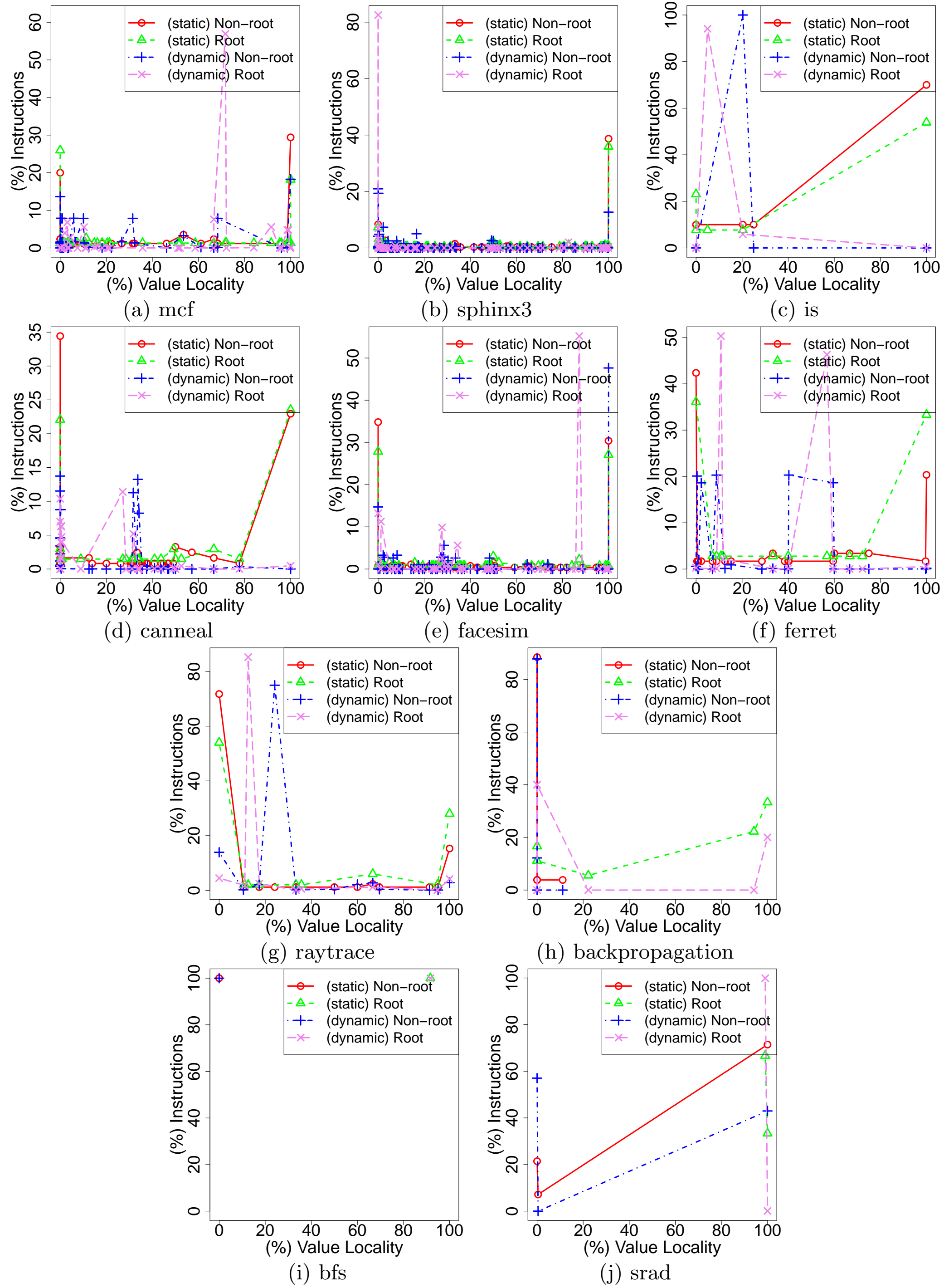

Figure 4.7: Value locality of *RSlice* instructions.

the locality of data values to be retrieved by the respective load instructions which are replaced by *RSlice*s.

Notice the distinction between static and dynamic instructions (for both root and non-root, i.e., intermediate instructions). Static instructions are the ones that are embedded in the binary by the compiler. Dynamic instructions are the ones that are actually executed at runtime. A static instruction may have multiple dynamic instances executed at runtime, or may not be executed at all. This distinction helps us to explain why, for instance, we do not obtain much benefit from **prediction** although a great fraction of static instructions have high value locality for *is* (Figure 4.7c): 53.84% of (static) root instructions of *is* have 100% value locality, but *is* does not benefit much from **prediction** (Figure 4.5). This is because, at runtime, the root instructions having 100% value locality are not executed as many times as other root instructions that have lower value locality. In fact, less than 1% of dynamic root instructions executed have 100% value locality for *is*, as shown in Figure 4.7c. The previous section revealed that bp benefits from **prediction** the most (Figure 4.5). Therefore, we expect a larger fraction of roots to have a very high value locality for this benchmark. Figure 4.7h reveals that 20% of dynamic root instructions of bp have 100% value locality indeed. A similar trend holds for non-root instructions under **recalculation+prediction**. For **recalculation+prediction**, prediction of (input operands of) non-root instructions can provide sizable gains only if the dynamic share of non-root instructions exhibiting (input operands of) high value locality is large.

Figure 4.8 shows how the node count of *RSlice*s change as the locality threshold to enable prediction increases from 50% to 100% under **recalculation+prediction** – *none* reflects no prediction, i.e., pure **recalculation**. The figure reports a histogram of node count of *RSlice*s (x-axis). The y-axis reports the % share of *RSlice*s having a given node count on the x-axis. A lower threshold enables more predictions, hence more producer instructions can get pruned, and the node count shrinks more. We observe that prediction at a value locality threshold of 50% can reduce the node count of *RSlice*s up to 56.5%. However, due to the limited value locality, this effect is barely visible for the majority of applications.

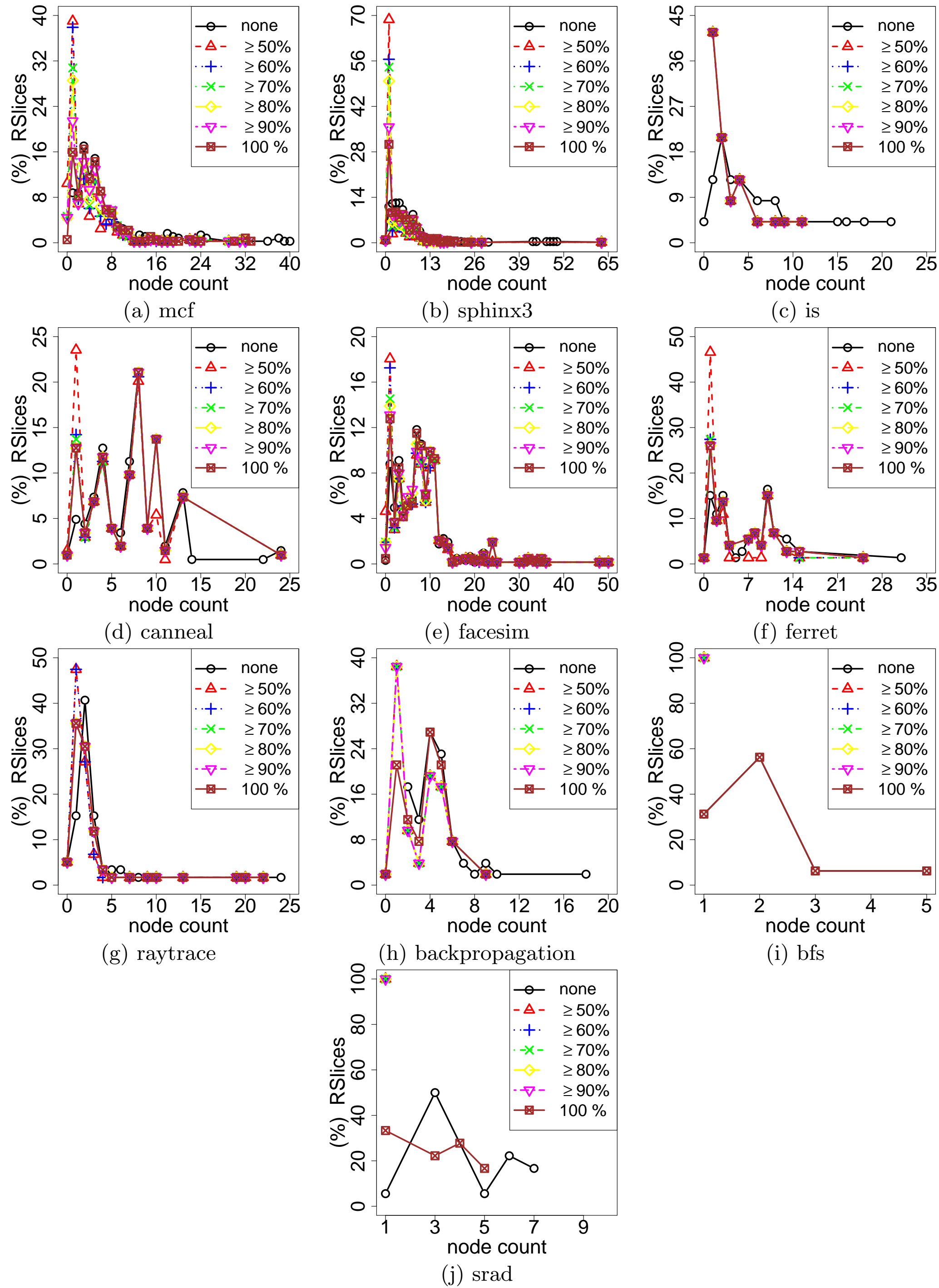

Figure 4.8: Node count of *RSlice*s before (**recalculation**) and after pruning (**recalculation+prediction**).

4.5 Summary

Recomputation can minimize, if not eliminate, the prevalent power and performance (hence, energy) overhead incurred by data storage, retrieval, and communication, thus, render more energy efficient execution. Recomputation replaces data load(s) from memory with the reproduction of the respective data. Unless the energy cost of reproducing data remains less than the energy cost of retrieving the same data from memory, recomputation cannot improve energy efficiency.

In this chapter, we explored (interactions between) two broad classes of recomputation techniques: brute-force recalculation and prediction based recomputation. Under recalculation, the recomputation effort goes to the generation of the data values (which would otherwise be loaded from memory), by re-executing the producer instruction(s) of the eliminated load(s). Under prediction, the recomputation effort goes to the estimation of the data values by exploiting value locality – the likelihood of the recurrence of values (which would otherwise be loaded from memory) within the course of execution. We find that recalculation has wider coverage for recomputation than prediction, as prediction cannot be effective under limited value locality.

Chapter 5

Recomputation-enabled Checkpointing and Recovery

5.1 Introduction

Scalable checkpointing is the key factor to enable emerging applications running on high-end computing platforms [49]. As we look into the such applications, we note that they need vast amount of processing capabilities, meaning more cores and associated components. Having more cores and using smaller feature sizes each technology generation result in higher probability of observing a fault during the lifetime of an execution. To ensure successful completion of an execution, a proper fault detection and recovery mechanisms have to be in place. The traditional method to recover from a fault is to periodically checkpoint the state of the program during its execution on reliable storage [50]. When a fault occurs, error-free consistent program state is constructed from the most recent checkpoint. The program is resumed after rolling back the execution to the most recent error-free consistent program state. Typically there are two main types of checkpointing and recovery mechanisms, namely coordinated [51] and uncoordinated [52]. The coordinated checkpointing and recovery has widely used since it is relatively simple, but incurs high overhead due to coordination with all the processes. Uncoordinated checkpointing and recovery, on the other hand, checkpoints without any coordination with others, so it provides maximum flexibility for processes to take checkpoints. However, it may not always be possible to find a consistent global

state to roll-back, making the (local) checkpoints useless; or it may require a chain of transitive rollbacks (a.k.a. domino effect [53]) which complicates the recovery process. In our discussions, we use global coordinated checkpointing and recovery (unless otherwise stated explicitly).

Checkpointing and recovery incurs a time overhead, t_{chk}, every time the program checkpoints, and a recovery overhead, t_{rec}, when the program restarts from the most recent checkpointed state after detection of a fault. The checkpoint overhead, t_{chk} is proportional to the time spent on recording the state, $t_{wr,chk}$, and the number of checkpoints, f_{chk}. The checkpoint overhead then becomes $t_{chk} = f_{chk} \times t_{wr,chk}$. The recovery overhead, t_{rec} includes the time (spent on useful work) lost since the last checkpoint, t_{waste} and the time spent on restoring the state of the last checkpoint, t_{rollb}. Therefore, under a fault rate of $perr$, the recovery overhead becomes $t_{rec} = perr \times (t_{waste} + t_{rollb})$.

In this chapter, we introduce a recomputation-enabled checkpointing to reduce the checkpointing overhead. Data recomputation can reduce both the frequency of checkpointing, and the size of the checkpoints, and thereby mitigate checkpointing overhead. The basic idea behind data recomputation is to eliminate the necessity of storing data to a checkpoint by relying on ability to recompute the desired value when it is needed (i.e. during recovery).

Under recomputation, time spent on recording the state, $t_{wr,chk}$ decreases since certain states (i.e. updated memory values) do not need to be checkpointed. This in turn decreases t_{chk}, even if f_{chk} remains the same. However, the recovery overhead t_{rec} now includes extra time spent on recomputing the states that are not checkpointed during the last checkpoint interval, t_{rcmp}. Still, the time spent on restoring the state of the last checkpoint, t_{rollb} is expected to decrease, since the size of checkpointed states is simply smaller. Therefore, the recovery overhead under recomputation becomes:

$$t_{rec,rcmp} = perr \times (t_{waste,rcmp} + t_{rollb,rcmp} + t_{rcmp})$$

To have $t_{rec,rcmp} \leq t_{rec}$, $(t_{rollb,rcmp} + t_{rcmp}) \leq t_{rollb}$ should be the case.

The primary contribution of this study is to analyze the impact of recomputation on checkpointing and recovery. Under recomputation, the checkpointing overhead can

be significantly reduced, while keeping the recovery overhead modest. We also devote a considerable discussion on how such a recomputation enabled microarchitecture can be designed and incorporated with checkpointing and recovery mechanisms. The proposed recomputation enabled checkpointing is:

- *hybrid (hardware/software)*: there is a need to generate a binary for recomputation enabled microarchitecture. Compiler has to extract the backward slices that would allow recomputation of data sets. At runtime, the microarchitecture has to identify the values that can be recomputed and should exclude them from consideration of checkpointing. In case of fault, such values have to be recomputed.

- *transparent*: Both recomputation enabled binary generation and facilitating recomputation on checkpoint and recovery are transparent to the application developer and user.

- *low overhead*: the main promise of recomputation is to mitigate the checkpointing overhead, while keeping the incurring costs of recomputation relatively low. There is a runtime overhead to identify the values that can be recomputed and to maintain the structures for supporting recomputation. Such costs are much lower than the benefits of recomputation.

- *scalable*: fundamentally, the checkpointing becomes challenging as the system or application scales. Recomputation mitigates the associated overheads of checkpointing (e.g. reducing the footprint and memory bandwidth requirements), making it more scalable.

5.2 Recomputation: Basic Idea

5.2.1 Support for Recomputation

In Chapter 3, we provide the details compiler and (micro)architecture support for opportunistic substitution of load instructions with arithmetic/logic instructions to recompute the data values which would otherwise be retrieved from the memory hierarchy. To facilitate recomputation-enabled checkpointing and recovery, we assume similar hardware-software support presented in Chapter 3.

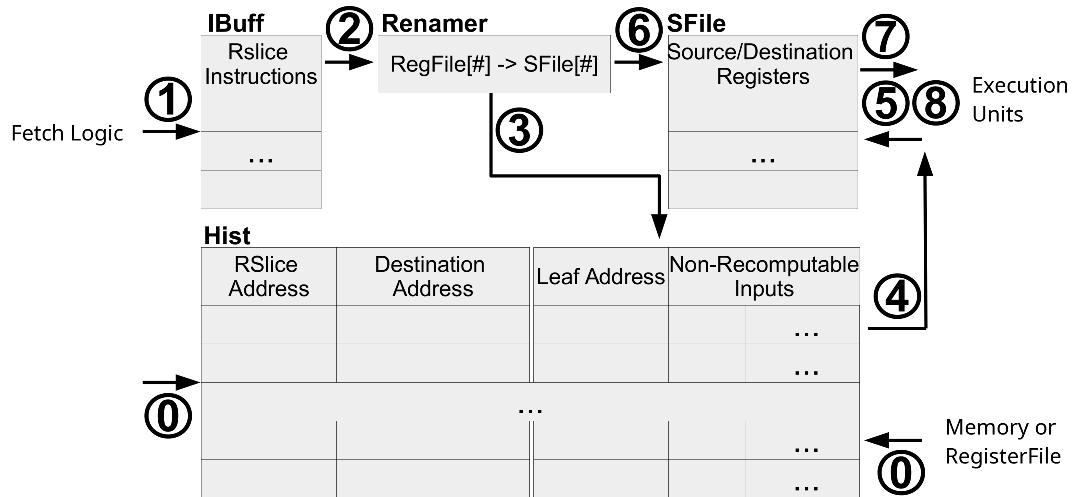

Figure 5.1: Microarchitectural support needed to facilitate recomputation.

Figure 5.1 captures microarchitectural support in orchestrating recomputation.

5.2.2 Recap: Recomputation Framework

The runtime scheduler tracks `RMCP` and `REC` instructions. `RCMP` instructs the scheduler to record the address of the *RSlice(v)* and the destination address of the value v into *Hist* (⓪ in Figure 5.1). Similarly, `REC` instructs the scheduler to record all non-recomputable input operands into *Hist* along with `leaf-address` (⓪).

A recomputation is triggered when certain events occur that set the `recompute` flag (e.g. detecting a fault and initiating a recovery). When `recompute` is set, the runtime scheduler goes over the *Hist*, fetches the address of *RSlice(v)*, and instruction fetch starts from there (which is the first leaf). Each leaf instruction first has its destination register renamed (② in Figure 5.1). Each leaf instruction with non-recomputable input operands next probes Hist with `leaf-address` (③) to read its input operands, which directly are fed into the corresponding execution units (④). Upon finishing execution, each leaf writes its result to the SFile (⑤).

Non-leaf recomputing instructions which represent intermediate nodes in *RSlice(v)* read their input operands from SFile (⑥) after having their source and destination registers renamed (②). Upon collecting the input operands, recomputing instructions

proceed to the execution units (⑦), and write their results back to the SFile once execution completes (⑧). All (non-leaf) recomputing instructions in *RSlice(v)* execute sequentially in this manner until the `RTN` instruction of the slice is fetched. Before return, the recomputed data value v gets copied from SFile into the destination address of v that was recorded by the `RCMP`. The runtime scheduler then resets `recompute` flag to demarcate the end of recomputation.

5.3 Checkpointing and Recovery

5.3.1 Checkpointing

A common approach for ensuring further progress and successful completion of an execution is to periodically checkpointing that is to save the state of an application to a reliable storage [50]. We focus on in-memory global checkpointing without loss of generality [54, 55, 51]; where all cores periodically cooperate to create a checkpoint, and we assume a reliable memory (see Section 5.3.2) as a storage for checkpoint. In-memory checkpointing has performance and power advantages over traditional checkpointing schemes that keep the checkpoint on disks. As the density and reliability measures of main memory enhance, we believe in-memory checkpointing will remain an appealing scheme. We use log-based incremental checkpointing that copies the old value at target address into a log, while the value is updated at target address [55, 51, 56]. The log is stored in-memory and contains the data that is needed to roll-back an error-free consistent state. The memory space can be reclaimed when a new checkpoint is established.

Global checkpointing is performed at regular periodic intervals. At the end of each period, all the cores are blocked and force to established a checkpoint. Establishing a checkpoint involves writing all dirty cache lines back to memory and copying the content of cores' architectural states (e.g. register file, status code register). The memory controller checks and copies the old value of a line in the memory into a log, before writing back the content of dirty cache line, if this is the first modification of the line since the last checkpoint. Similar to [55] a modified cache line is required to be logged only once between a pair of checkpoints. To facilitate this, the directory controller has an additional bit for each memory line, and this bit is used to determine whether a particular line has already been logged for the current checkpoint interval. The bit for

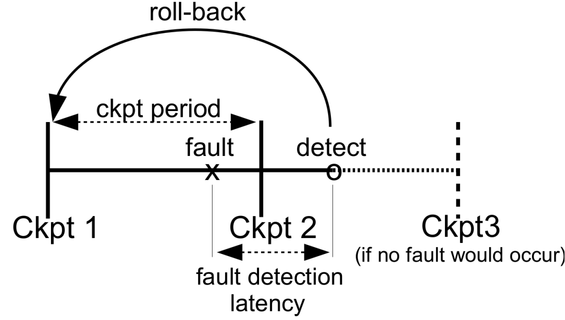

Figure 5.2: Recovery from a fault.

a line is set when the line is logged, and cleared when a new checkpoint is established.

5.3.2 Error Detection and Recovery

Without loss of generality, we assume a fail-stop error model and error detection. Also, we assume that data memory and checkpoint logs do not suffer from any faults, similar to [56]. This can be ensured through various mechanisms, such as ECC [57], non-volatile memory, or memory raiding [58]. To detect the errors, the system can use modular redundancy [59], or error detection codes (e.g. CRC). Further specifications and assumptions on error detection is beyond the scope of this study.

Error detection is not instantaneous, meaning there is a lag between the occurrence of an error and its detection, referred as error detection latency. As a consequence, more than one checkpoints have to be stored before one is validated as being error-free. We assume that the fault detection latency is no longer than a checkpoint period, so we have to keep the most recent two checkpoints to ensure that we have an error-free checkpoint in case of fault. Figure 5.2 illustrates the need for keeping the most recent two checkpoints and how to recovery from a fault. The time passed between the recently established checkpoint and the error detection is less than error detection latency, so there is no guarantee for recently established checkpoint (*Ckpt2*) to be error-free. To recover from the fault in such a case, we should have the second most recent checkpoint at hand (*Ckpt1*). After rebuilding the global consistent recovery line by restoring the states kept in the *Ckpt1*, the execution can resume starting from this point onward.

5.4 Incorporating Recomputation in Checkpointing and Recovery

In this section, we detail how data recomputation can be exploited to reduce the checkpointing overhead. Data recomputation can reduce both the frequency of checkpointing, and the size of the checkpoints by eliminating the necessity of storing data to a checkpoint by relying on ability to regenerate the desired value when it is needed (i.e. during recovery). We focus on in-memory global checkpointing without loss of generality; however, recomputation can similarly reduce the overhead of other checkpointing schemes.

First, we discuss how data recomputation support can be exploited and orchestrated with checkpointing mechanism. Then, we explain the necessary actions of recovery process in recomputation enabled checkpointing when the fault is detected.

5.4.1 Recomputation Enabled Checkpointing

Assuming a compiler introduced in Section 3.3.1 generates a binary that is annotated and contains all the viable *RSlice*s. The runtime scheduler, then records the address of the *RSlice*s and the destination addresses for value to be recomputed into *Hist* when it encounters `RCMP` instruction (see Section 3.3.2). Similarly, it records non-recomputable input operands of *RSlice*s when `REC` instruction is encountered, making sure all the necessary inputs for a given *RSlice* are available. When runtime scheduler records a particular *RSlice* into *Hist* it also request memory controller to set the bit used for deciding if a given value should be logged (see Section 5.3.1). The semantic of setting this bit is letting memory controller know that the given value v has corresponding *RSlice(v)* and it can be recomputed when it is needed (i.e. in recovery). When memory controller receives such a request, it sets the bit and excludes the value v from the consideration of storing it into a checkpoint for that interval. Eventually, the size of checkpoint reduces as more values have *RSlice*s.

The *RSlice*s and the input operands have to remain in *Hist* as long as the established checkpoint for the given interval is stored in memory. In case of a fault, the global state must be restored in coordination with the established checkpoint and *RSlice*s in *Hist*. The *RSlice*s will be used to recompute the values that were not checkpointed. Assuming fault detection latency being no more than checkpointing period, we should retain the

most recent two checkpoints (see Section 5.3.2); similarly, *Hist* should retain the *RSlice*s and input operands for the most recent two checkpoints.

For each checkpointing interval, we can eliminate as many values from checkpointing as the maximum number of RSlices that can fit into the *Hist*. Once the history buffer runs out of space, all the values have to be checkpointed even if they have a corresponding *RSlice*. For more detailed discussion on size of *Hist*, please see Section 5.4.3.

5.4.2 Recomputation Enabled Recovery

When a fault is detected, the most recent error-free consistent global recovery line should be built by restoring the checkpoint. Under recomputation enabled checkpointing, there might be values that are not checkpointed and can not be found in the checkpoint. To build the global recovery line, these omitted values have to be recomputed. Although the checkpoint does not have these values, the corresponding *RSlice*s are present in the *Hist*. These missing values will be recomputed as explained in Section 5.2.2. When a fault is detected the `recompute` flag is set and the runtime scheduler goes over the *Hist*, fetches the corresponding *RSlice(v)* and starts to schedule its instructions to execute. Each *RSlice(v)* generates the missing value v, and then it is stored back to the destination address which is recorded in *Hist* as well. Note that there is no need to maintain a separate bookkeeping for the values missing from checkpoint, since *Hist* records the corresponding *RSlice*s.

After recomputing the missing values and storing them back to their destination addresses, the rest of the states in checkpoint can be restored. At the end, the whole states necessary to establish a global recovery line are restored, and execution then resumes starting from this point onward.

5.4.3 Microarchitecture Support for Recomputation-Enabled Checkpointing

To exploit the potential of recomputation-enabled checkpointing, the underlying microarchitecture should provide the necessary support introduced in Section 5.2.1. Similar to data memory and checkpoints, we assume *Hist* does not suffer from any fault. Such a protection can be obtained through ECC (for further discussion, see Section 5.3.2).

The *RSlice*s cannot grow indefinitely, as the overhead of recomputation increases with the size (in terms of the number of instructions) of *RSlice*s. The performance and energy overhead of recomputation can easily outweigh the benefits if *RSlice*s become excessively large. To prevent this, a threshold can be set for the maximum number of instructions per *RSlice*, which the compiler takes into account this threshold to filter in embedding *RSlice*s into the binary.

The memory controller should be extended, similar to [55], to maintain a bit for determining if the old value of a given write-back should be logged. For each write-back request, the memory controller has to determine (i) whether the request would result in the first update to the respective memory line since the last checkpoint was taken, and (ii) whether the current data value v of the respective memory line (i.e., the value before the write-back takes place) can be recomputed. While memory controller can maintain the bit itself for (i), it should cooperative with the runtime scheduler for (ii). The runtime scheduler should send a request to memory controller and let it know the given value v can be recomputed, so it should not be checkpointed. The memory controller sets the bit when it receives the request from runtime scheduler.

The number of (stores corresponding to the) values that can be excluded from checkpointing depends on the size of the *Hist*, i.e., how many *RSlice*s the *Hist* can keep track of. Fortunately, we do not need to have an excessively large *Hist* to this end: Recall that we only need to checkpoint the old values on the very first write-backs (to unique addresses) when a new checkpoint is established. Therefore, the number of *RSlice*s is not a function of how many times an address is updated, but *how many unique memory address is updated* within a given checkpoint interval. Naturally, the latter is bounded by the period of checkpointing. As the period gets longer, the probability of having a higher number of unique memory addresses updated increases. At the same time, as the period gets longer, the amount of useful work lost upon detection of a fault increases. The checkpointing period cannot get too long to reduce this amount of useful work lost. The checkpointing period hence puts an upper bound on how many unique RSlices can be encountered at runtime.

5.4.4 Overheads

There is a performance overhead of establishing a checkpoint. When it is time to checkpoint, all the cores have to be blocked and all the dirty cache lines have to write-back to the memory. Before updating the lines in memory, the existing values have to be logged for checkpoint (if this is the first write-back, see Section 5.3.1 for further discussion). Recomputation can reduce the amount of values to be logged for checkpoint, so the performance overhead of checkpointing is likely to reduce. On the other hand, *RSlice*s to recompute the missing values in checkpoint have to be recorded in *Hist* that can be performed in parallel to the other operations. So, *Hist* update latency can be hidden. Although, the update latency can be hidden, there is energy overhead of updating *Hist*.

The size of checkpoint (i.e. storage overhead) reduces under recomputation enabled checkpointing since the number of values checkpointed shrinks. Such a reduction in checkpoint size can be reflected on energy saving as well as performance gain due to less amount of memory read/write operations (for recovery and checkpoint, respectively).

When an error is detected, all cores have to be blocked and roll-back and recovery should be initiated. Recovery includes the recomputation of missing values from the checkpoint and restoring the rest of the states in checkpoint. Recomputation incurs a performance overhead; however,it is not prohibitive since the number of instructions in the *RSlice*s are bounded (see Section 5.4.3). Although recomputation introduces extra overhead for recovery, it reduces the number of values to be restored. The performance benefit of having smaller set of values to be restored may or may not be comparable to the overhead of recomputation. However, considering the number of checkpoints and the number of recoveries, one can argue that recovery is more likely to be less frequent event compared to checkpoint, so the performance benefits of recomputation overweight its overhead (due to recovery).

Another overhead associated with the recovery is to re-perform the work that has been lost. This overhead remains the same for traditional checkpointing and recomputation enhanced checkpointing.

Technology node:	22nm
Operating frequency:	1.09 GHz
4-issue, in-order, 8 outstanding ld/st	
L1-I (LRU):	32KB, 4-way, 3.66ns
L1-D (LRU, WB):	32KB, 8-way, 3.66ns
L2 (LRU, WB):	512KB, 8-way, 24.77ns
Main Memory	120ns, 7.6 GB/s/controller 1 mem. contr. per 4-cores
Network Bandwidth	128 GB/s

Table 5.1: Simulated architecture to evaluate the impact of recomputation on checkpointing and recovery.

5.5 Evaluation Setup

To evaluate the impact of recomputation on checkpointing and recovery, we experiment with eight benchmarks (excluding ep due to technical difficulties on running it on simulation environment) from NAS [11] suite. We relied on OpenMP version of NAS benchmarks for parallel implementation. We run the benchmarks with 8 threads on simulated 8-core system.

We based our simulations on a similar configuration presented in Section 3.4: (Intel's Xeon Phi like) an in-order core, running at 1.09GHz, with a private L1 and shared L2 cache. We extended Sniper-6.1 [17] to facilitate recomputation, as well as checkpointing and recovery mechanism we propose. The energy measurements are extracted from McPAT [16] that is integrated with the Snipersim. Table 5.1 summarizes the main configuration of the core and the system.

We implemented the greedy compiler pass to generate a recomputation-enabled binary as a Pin [14] tool, similar to presented in 3.4. however, instead of using probabilistic energy-per-instruction cost of memory accesses to filter out the *RSlice*s, we used a predetermined threshold for *RSlice* length: the *RSlice*s exceeding the given threshold are not included into binary.

For the evaluation, we used a baseline that we assume fault-free execution without

any checkpointing (*No_Ckpt*). Then, we modeled two configurations for global checkpointing: i) periodic checkpointing, fault-free execution (*Ckpt_NF*); ii) periodic checkpointing, fault-incurred execution (*Ckpt_F*). In *Ckpt_NF* configuration, we modeled the mechanism of coordinated global checkpointing where all cores are halt and checkpoint their respective architectural and memory states at regular periodic intervals. In this configuration, we assume there is no fault, so we can clearly see the overhead of global checkpointing. In *Ckpt_F* configuration, in addition to coordinated global checkpointing, we also modeled the mechanism of recovery when a fault is detected.

To characterize the impact of recomputation enabled checkpointing and recovery, we also modeled the following two configurations: i) recomputation enabled periodic checkpointing, fault-free execution (*Rec_Ckpt_NF*); ii) recomputation enabled periodic checkpointing, fault-incurred execution (*Rec_Ckpt_F*). In *Rec_Ckpt_NF* configuration, we incorporated recomputation into the global checkpointing where the size of checkpoint can be reduced due to the eliminated values that can be recomputed in case of a need (i.e. in recovery). Since we assume fault-free execution for this configuration, we can clearly see the impact of recomputation on checkpoint size and its governing overheads. In *Rec_Ckpt_F* configuration, we modeled the recomputation enabled recovery, in addition to checkpointing. We assume certain number of faults occurs during the execution and all the cores have to recovery when a fault occurs. During the recovery process, first, the omitted states that are necessary to establish global recovery line are recomputed, and then the remaining states are restored from the checkpoint. The configurations that are modeled are summarized in Table 5.2. We set the checkpointing frequency for benchmarks to checkpoint 100 times for benchmarks where execution takes longer (i.e. bt, cg, lu, and sp), and 25 times for benchmarks where execution takes relatively shorter (i.e. dc, ft, is, and mg). The checkpoint intervals are uniformly distributed over the execution time. We remind that *No_Ckpt* does not involve checkpointing, so it is overhead-free baseline. We assume a fault occurs during the execution and all the cores have to recovery when a fault occurs for *Ckpt_F* and *Rec_Ckpt_F* configurations under global coordinated checkpointing.

Configuration	Explanation
No_Ckpt	the baseline assuming fault-free execution, no checkpointing
Ckpt_NF	periodic checkpointing, assuming fault-free execution
Ckpt_F	periodic checkpointing, assuming fault occurs in execution
Rec_Ckpt_NF	recomputation enabled checkpointing, assuming fault-free execution
Rec_Ckpt_F	recomputation enabled checkpointing, assuming fault occurs in execution

Table 5.2: The summary of configurations evaluated.

5.6 Evaluation

5.6.1 Checkpointing Overhead in Fault-Free Execution

In this section, we want to present the impact of recomputation on performance, energy and energy-delay product –as a proxy for energy efficiency– (EDP [18]) of a fault-free execution. We use *No_Ckpt* as baseline where no checkpointing takes places. Figure 5.3 shows the normalized execution time of benchmarks under *Ckpt_NF* and *Rec_Ckpt_NF* configurations. The general trend is that *Ckpt_NF* and *Rec_Ckpt_NF* have consistently worse performance compared to *No_Ckpt* due to the checkpointing overhead. Notice that *Rec_Ckpt_NF* is very effective in reducing the performance overhead of checkpointing involved in *Ckpt_NF* *Rec_Ckpt_NF* reduces the performance overhead of *Ckpt_NF* up to 28.81% (for is), and 11.92%, on average. The smallest reduction is 2.12% for cg. This small reduction is due to the fact that, in cg, the performance overhead of *Ckpt_NF* is also relatively low. This is because cg has relatively long execution time and the checkpoint size per checkpoint interval is relatively small; the amount of time spent in checkpointing accounts $\approx 9\%$ of total execution time.

Figure 5.4 shows the normalized system energy of benchmarks under *Ckpt_NF* and *Rec_Ckpt_NF* configurations. General trend is similar to performance and *Rec_Ckpt_NF* reduces the energy overhead of checkpointing involved in *Ckpt_NF*. *Rec_Ckpt_NF*reduces the energy overhead of *Ckpt_NF* up to 26.93% (for is), and 12.53%, on average. In Section 3.5.1, we see that *is* benchmark is very amenable to recomputation. Thus, here as well, we see that majority of the updates to memory can be recomputed in a cost effective manner (in case of recovery), so they can be excluded from checkpoint set, providing higher reduction in overhead associated with checkpointing in *Ckpt_NF* The smallest energy reduction is 1.75% (for cg) due to the reasoning provided in performance

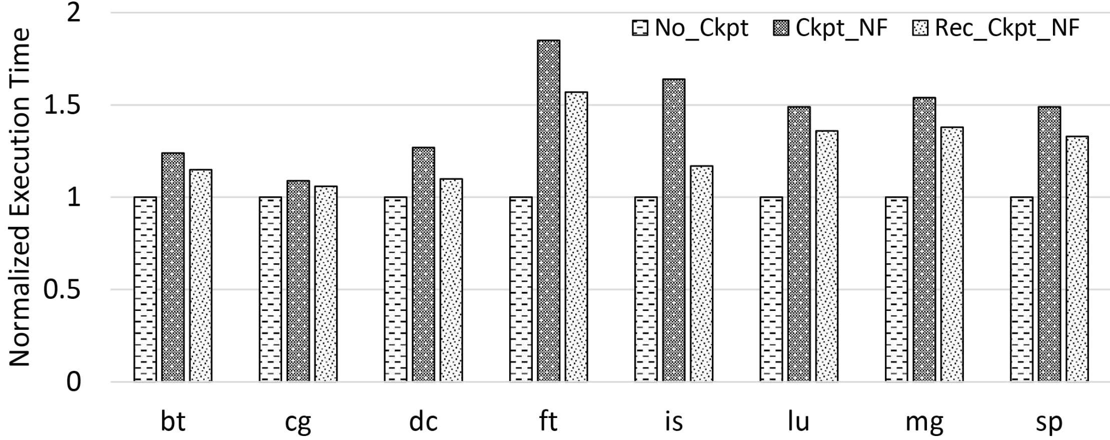

Figure 5.3: Normalized execution time of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_NF* and *Rec_Ckpt_NF* configurations.

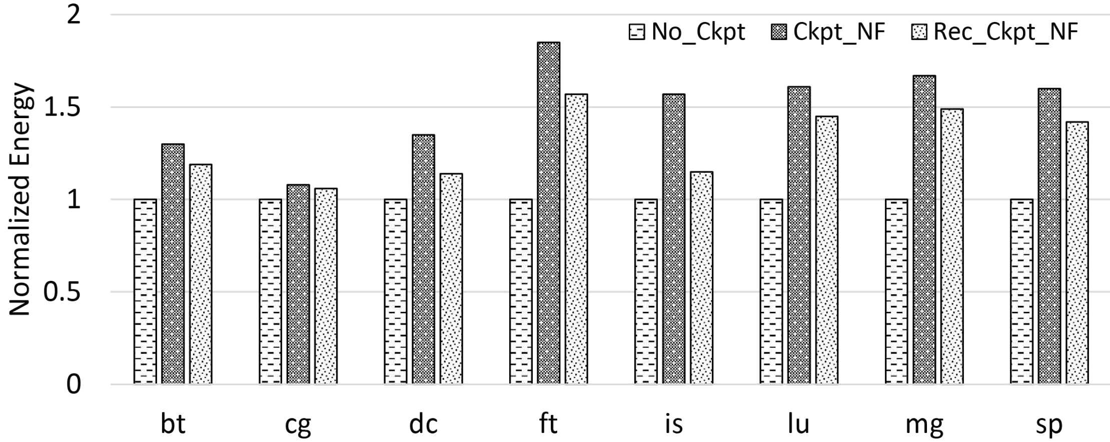

Figure 5.4: Normalized energy consumption of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_NF* and *Rec_Ckpt_NF* configurations.

overhead discussion.

Figure 5.5 shows the normalized energy-delay product (EDP) of benchmarks under *Ckpt_NF* and *Rec_Ckpt_NF* configurations. We use EDP as metric for evaluating the energy efficiency of the recomputation-enabled checkpointing. Generally, EDP provides a notion of balance between the performance overhead and energy consumption. *Rec_Ckpt_NF* provides EDP gain up to 47.98% (for is), and 22.47%, on average.

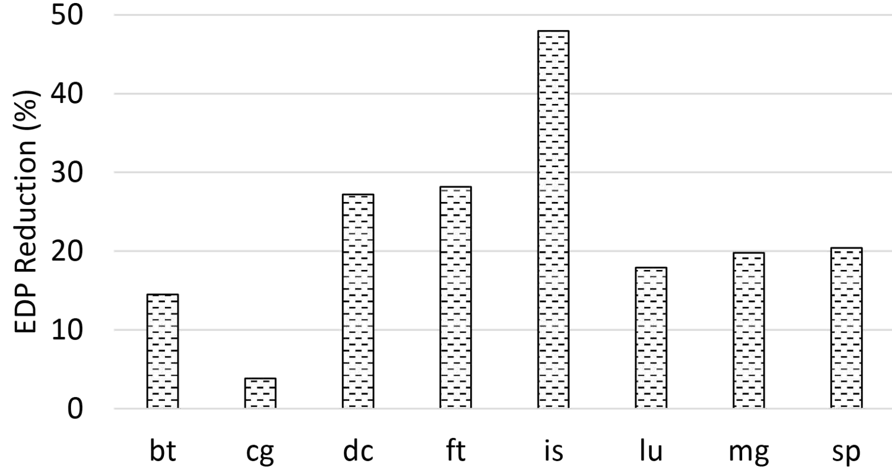

Figure 5.5: EDP reduction of benchmarks under *Rec_Ckpt_NF* configuration (w.r.t. *Ckpt_NF*).

5.6.2 Recovery Overhead in Fault-Occurring Execution

In Section 5.6.1, we assumed no fault occurs during the course of execution; however, we regularly checkpoint to quantify the pure overhead of checkpointing. In this section, we want to quantify the overhead of recovery process, in case of a fault occurs in execution. Recovery requires to establish a globally consistent state among all threads. For *Ckpt_F* configuration that means each thread has to rollback and restore the states kept in most recently established checkpoint. On the other hand, *Rec_Ckpt_F* configuration requires each thread to restore the states kept in most recently established checkpoint, as well as to recompute the values that have been omitted during the time of establishing of checkpoint. Such values were omitted during checkpointing since they have corresponding *RSlice*s which can be triggered to recompute them at a later time. Thus, although *Rec_Ckpt_F* configuration reduces the checkpointing overhead, it requires extra effort to recompute the missing states. To avoid excessive overhead due to recomputation, we only select the *RSlice*s that have at most certain number of instruction (by doing so, we put a cap on the recomputation overhead). For the discussion in this section, we select the *RSlice*s having at most 10 instruction. For the sensitivity analysis on *RSlice* length, please refer to Section 5.6.8.

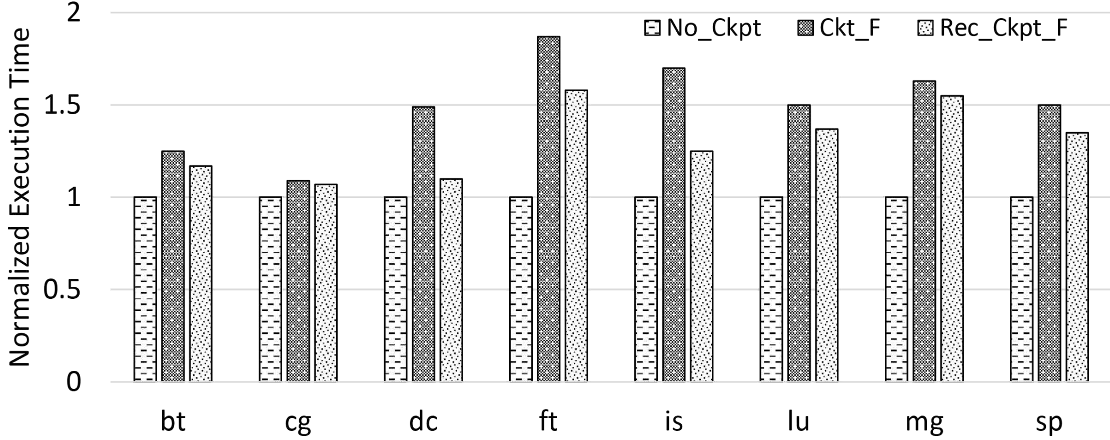

Figure 5.6: Normalized execution time of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_F* and *Rec_Ckpt_F* configurations.

Similar to analysis in Section 5.6.1, we use *No_Ckpt* as baseline where no checkpointing takes places (and we still assume no fault occurs in *No_Ckpt*). Figure 5.6 shows the normalized execution time of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations, where we a fault occurs during the execution. The performance overheads of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations are higher than *Ckpt_NF* and *Rec_Ckpt_NF* respectively. This is because, in addition to checkpointing overhead, *Ckpt_F* and *Rec_Ckpt_F* include the recovery overhead. *Rec_Ckpt_F* is very effective in reducing the performance overhead of *Ckpt_F*Ȧlthough *Rec_Ckpt_F* needs to recompute the missing values, thus incurs additional overhead, reduction of checkpointing overhead (due to the reduced checkpoint size) and reduction of the restore overhead (again, due to the reduced checkpoint size) outweighs the associated overhead of recomputation. For this reason, *Rec_Ckpt_F* provides a low-cost checkpoint and recovery.

Rec_Ckpt_F reduces the performance overhead of *Ckpt_F* up to 26.68% (for is), and 12.39%, on average. The smallest reduction is 1.9% for cg. Similar to the previous justification on fault-free execution, this small reduction is due to the fact that, in cg, the performance overhead of *Ckpt_F* is also relatively low.

Figure 5.7 shows the normalized system energy of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations. The energy reduction follows the very same trend with the performance overhead reduction. *Rec_Ckpt_F* reduces the energy overhead of *Ckpt_F* up

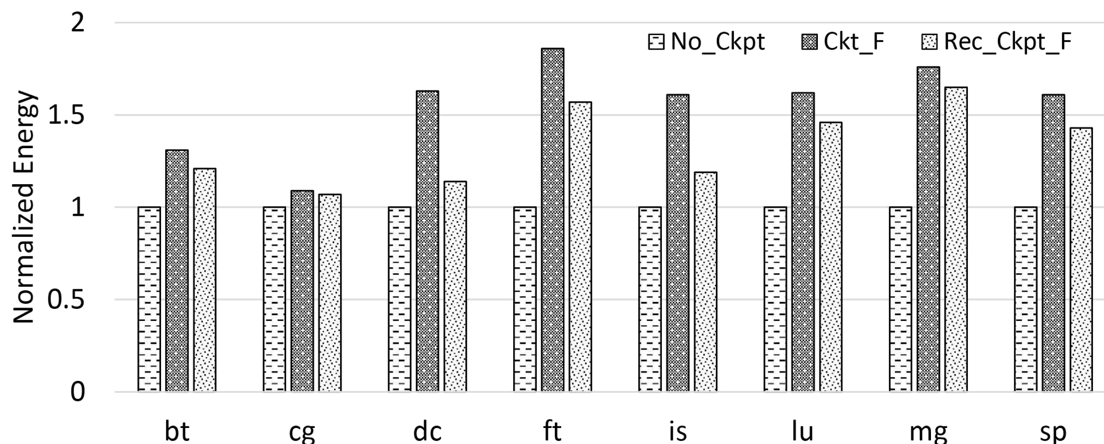

Figure 5.7: Normalized energy consumption of benchmarks (w.r.t. *No_Ckpt*) under *Ckpt_F* and *Rec_Ckpt_F* configurations.

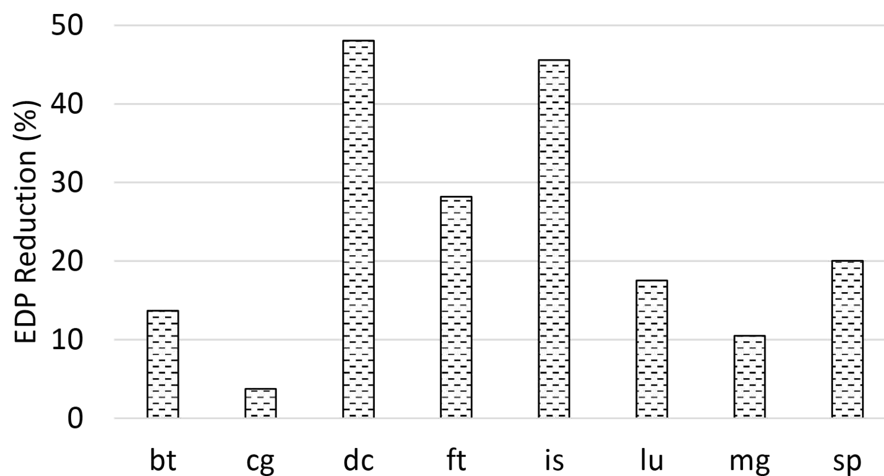

Figure 5.8: EDP reduction of benchmarks under *Rec_Ckpt_F* configuration (w.r.t. *Ckpt_F*).

to 30% (for dc), and 13.47%, on average. The smallest energy reduction is 1.86% (for cg).

Finally, Figure 5.8 shows the normalized energy-delay product (EDP) of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations. *Rec_Ckpt_F* provides EDP gain up to 48.07% (for dc), and 23.41%, on average. Notice that although is benchmark benefits more from *Rec_Ckpt_F* in terms of performance, dc benchmark has higher energy gain due to *Rec_Ckpt_F*; and in turn dc has higher EDP gain.

Data recomputation effectively reduces the associated costs of checkpointing, as well as rollback and recovery. The effectiveness of recomputation-enabled checkpointing highly depends on the low-cost *RSlice*s and how many values can be excluded from checkpoint. The analysis of the impact of *RSlice* length on checkpoint size reduction is presented in Section 5.6.8.

5.6.3 Checkpoint and Footprint Size Reduction

The recomputation-enabled checkpointing demonstrates big potential for mitigating the checkpointing overhead, due to its promise of reducing the amount of data to be checkpointing. The reduction of checkpoint size has mainly two implications. The first one is the amount of data to be moved to designated memory area is reduced; thus saving energy and reduces time required to perform copy. Second, the size of a particular checkpoint is shrunk, so the footprint of a checkpoint on memory (i.e. required memory size) can also be reduced. The largest checkpoint among all checkpoints (i.e. maximum size) designates the memory footprint of the checkpoint (we assume the memory space allocated to previous checkpoints can be reclaimed). Since we need to keep two most recent checkpoints (see Section 5.3.2 for details), the memory space we have to allocate for checkpoints is the 2× size of the largest checkpoint. As recomputation-enabled checkpointing can shrink the size of checkpoint, the memory footprint (i.e. required memory space) may also be shrunk. Such shrinkage on memory requirement may lead to extra energy benefits (e.g. due to less leakage and refresh in case of DRAM).

Figure 5.9 shows the percentage of total checkpoint size reduction under *Rec_Ckpt_NF* (w.r.t. to *Ckpt_NF*). Among all the benchmarks, is benefits the most from recomputation, and total checkpoint size is reduced by 75.74% under *Rec_Ckpt_NF*. On the other hand, the benefits are limited for cg, having total checkpoint size reduction by 6.99% under *Rec_Ckpt_NF*. The average checkpoint size reduction is 38.31% for the benchmarks under under *Rec_Ckpt_NF*. The reductions for *Rec_Ckpt_F* are inlined with the *Rec_Ckpt_NF* (since having a fault does not change the set of values that can be recomputed and set of values to be checkpointed).

On the other hand, Figure 5.10 shows the percentage of footprint size reduction under *Rec_Ckpt_NF* (w.r.t. to *Ckpt_NF*). Notice that, recomputation-enabled checkpointing can reduce the memory footprint size, if it reduces the size of the checkpoint

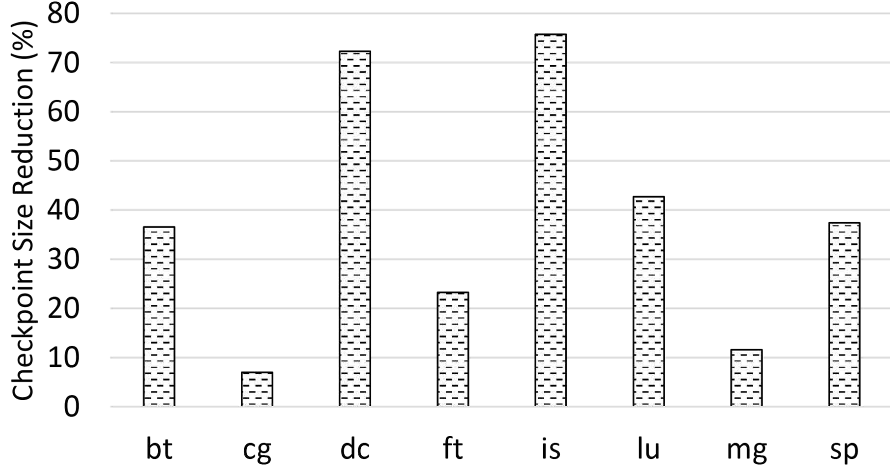

Figure 5.9: Percentage of checkpoint size reduction under *Rec_Ckpt_NF* configuration.

that is the largest among all checkpoints. If there is no value that can be recomputed within the largest sized checkpoint, then recomputation can not reduce the footprint size; although it may reduce the total amount of data to be checkpointed. Such a case can be seen in Figure 5.10. Among the benchmarks is has very limited footprint reduction (2.04%) under *Rec_Ckpt_NF*; although it has the highest checkpoint size reduction (see Figure 5.9). For the rest of the benchmarks, dc has the largest footprint size reduction that is 58.3%, and ft has the smallest footprint size reduction that is 0.05%. For ft, this means *Rec_Ckpt_NF* can reduce the size of largest checkpoint by only 0.05%, while it can reduce the total checkpoint size by 23.27% (see Figure 5.9). Similar to checkpoint size reductions, the footprint size reduction for *Rec_Ckpt_F* are inlined with the *Rec_Ckpt_NF* (due to the same argument: a fault does not change the set of values that can be recomputed and set of values to be checkpointed).

5.6.4 Impact of Thread Count on Checkpointing Overhead

One factor that directly impacts the overhead of checkpointing is the number of threads involved in execution. As the number of threads increases, the associated costs of checkpointing also increases. First of all, the coordination burden among threads to checkpoint and the amount of states to be checkpointing increases. As a consequence the memory bandwidth requirement also increases as multiple threads need to access

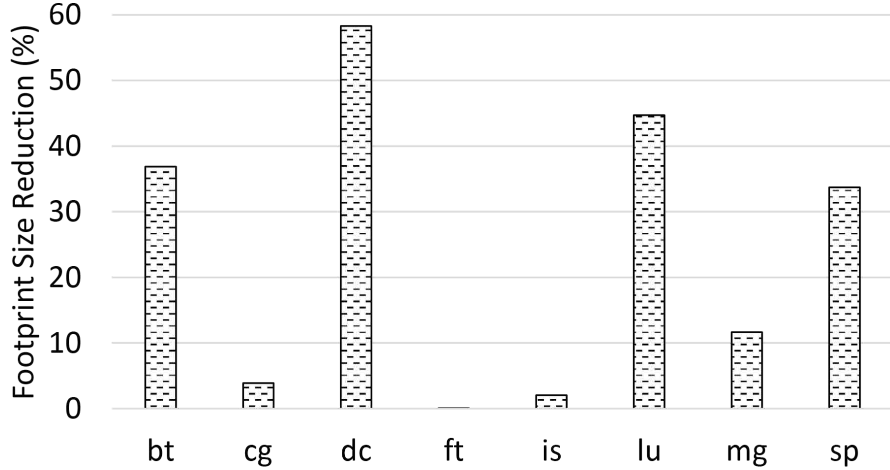

Figure 5.10: Percentage of footprint size reduction under *Rec_Ckpt_NF* configuration.

memory to complete the checkpoint. Recomputation-enabled checkpointing alleviates the overhead of checkpointing and remain effective as the number of threads scales up. To evaluate the effectiveness of recomputation-enabled checkpointing, we experimented with 8-,16-, and 32-threaded executions for the given benchmarks (we increase the core count as we increase the thread count: each thread maps to a separate core).

Figure 5.11 shows the performance overhead of checkpointing under *Ckpt_NF* configuration, as we increase the thread count from 8 to 32. The bars indicate the performance overhead of *Ckpt_NF* configuration for a given thread count compared to performance of *No_Ckpt* for that thread count. As an example, the bar shown as 8-thread under bt indicates the performance overhead of *Ckpt_NF* running with 8 threads w.r.t. *No_Ckpt* running with 8 threads. Similarly, the bar shown as 16-thread under bt indicates the performance overhead of *Ckpt_NF* running with 16 threads w.r.t. *No_Ckpt* running with 16 threads. Although there is no specific pattern, checkpointing overhead always remains more than 9% for any thread count. On average, the checkpointing overhead is $\approx$ 45%, 55%, and 60% for 8-, 16-, and 32-threaded executions, respectively, under *Ckpt_NF* configuration.

Figure 5.11 makes it clear that the checkpointing overhead is considerable regardless of thread count which motivates us further to exploit data recomputation for reducing checkpointing overhead. Figure 5.12 shows the percentage of performance overhead reduction when benchmarks running with 8-, 16-, and 32-threads under *Rec_Ckpt_NF*.

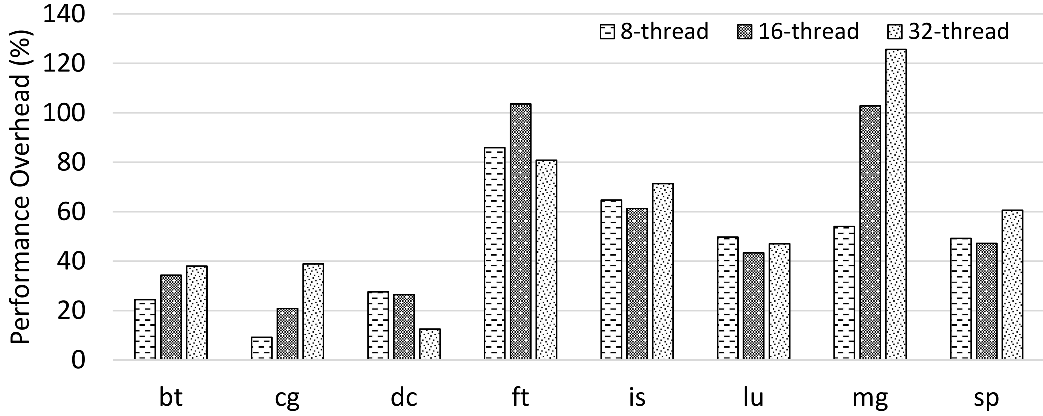

Figure 5.11: Performance overhead of checkpointing for 8- 16- and 32-threaded executions under *Ckpt_NF* configuration.

The performance overhead is reduced up to 28.81% (for is), 17.78% (for is), and 19.12% (for mg) when running with 8-, 16-, and 32-threads, respectively, under *Rec_Ckpt_NF*. Average performance overhead reduction is ≈12% for 8-threaded executions, and ≈11% for 16- and 32-threaded executions.

In addition to performance overhead reduction, recomputation-enabled checkpointing reduces the energy overhead as well, resulting better EDP for the benchmarks. Under *Rec_Ckpt_NF* configuration, the EDP improves up to 47.98% (for is), 31.81% (for dc), and 33.8% (for mg) when running with 8-, 16-, and 32-threads, respectively. Average EDP improvement under *Rec_Ckpt_NF* configuration is ≈22%, 21% and 20% for 8-, 16-, and 32-threaded executions.

The performance overhead reduction and EDP improvements under *Rec_Ckpt_F* configuration closely follow the *Rec_Ckpt_NF* for 8-, 16-, and 32-threaded executions.

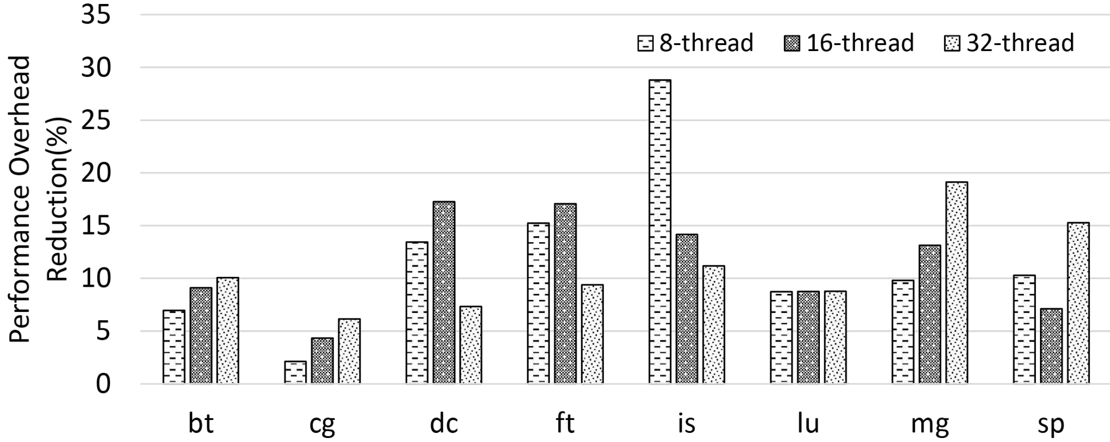

Figure 5.12: Performance overhead reduction of checkpointing for benchmarks running with 8-, 16-, and 32-threads under *Rec_Ckpt_NF* configuration.

5.6.5 Impact of Fault Rate on Recovery Overhead

The fault rate directly dictates the rollback and recovery overhead. As more faults occur, more overhead incurs in execution. For the analysis we have shown so far for *Ckpt_F* and *Rec_Ckpt_F* we assume a single fault occur during the course of execution. In this section, we want to extend the analysis for multiple faults and evaluate the overhead reduction promise of data recomputation.

As a reminder, the overhead of recovery under *Rec_Ckpt_F* configuration includes the cost of recomputing the missing values that were not in the set of values to be checkpointed (they are omitted since they can be recomputed). When a fault occurs, the missing values will be recomputed and then restored. Overall, *Rec_Ckpt_F* should reduce the recovery overhead if the restore overhead reduction due to smaller checkpoint size dominates the cost of recomputing the missing values. This is highly dependent on the cost of corresponding *RSlice*s of missing values. The cost of *RSlice*s can not grow indefinitely, since we have an upper limit on the number of instructions that an *RSlice* can have. We exclude the *RSlice*s from consideration (at the time of binary generation), if the length of (i.e. number of instructions) *RSlice* exceeds the limit. Thus, we make sure that the recomputation overhead remains reasonably low and does not exceed the recovery cost of *Ckpt_F*.

If the fault rate increases (i.e. number of faults occur in execution), we expect to

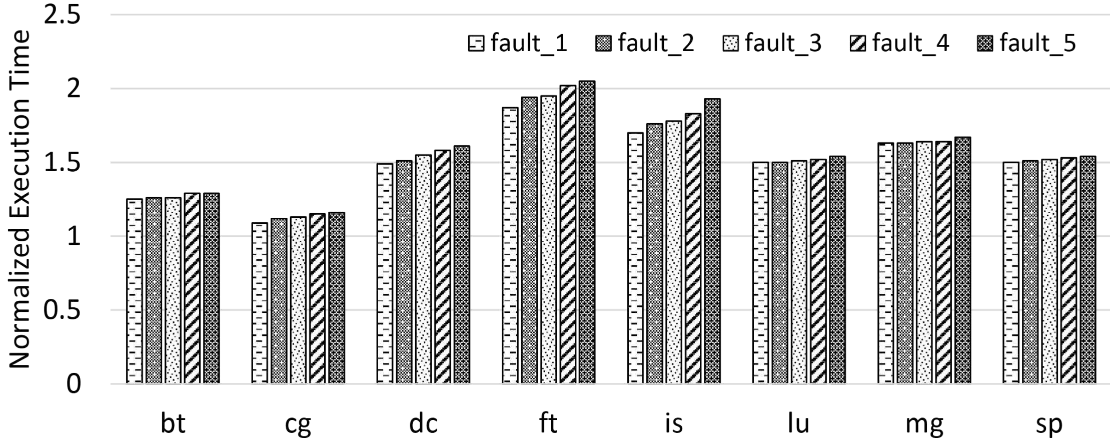

Figure 5.13: Normalized execution time under *Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates.

have accumulated overhead due to multiple recoveries needed. Figure 5.13 shows the normalized execution time under *Ckpt_F* (w.r.t. *No_Ckpt*) as the fault rate varies. In our evaluation, we change sweep the fault rate in a way that the total number of faults occur in execution range between 1 and 5. We uniformly distribute these faults within execution in our evaluations. In Figure 5.13, the number of faults corresponding to different fault rates are labeled as fault_1 for a single fault, fault_2 for two faults occur during the course of execution, and so on. Not surprisingly, the execution time increases as the fault rate increases. Some benchmarks experience higher performance overhead as the fault rate increases. This is mainly because the execution time under *No_Ckpt* is relatively small, and the overhead of rollback and recovery proportionally higher. Among the benchmarks, ft suffers the most as its per recovery overhead is relatively high.

Figure 5.14 shows the normalized execution time under *Rec_Ckpt_F* (w.r.t. *No_Ckpt*) as the fault rate changes. While the pattern is very similar to *Ckpt_F* configuration, the overheads are lower, since overall recovery overhead (including restore the checkpointed values, and recomputing missing values) is considerably low under *Rec_Ckpt_F* configuration. The performance overhead is reduced up to 26.68% (for is) for single fault, 25.35% (for dc) for two faults, 26.87% (for dc) for three faults, 21.58% (for dc) for four faults, and 19.92% (for is) four five faults occur in execution under *Rec_Ckpt_F* (w.r.t.

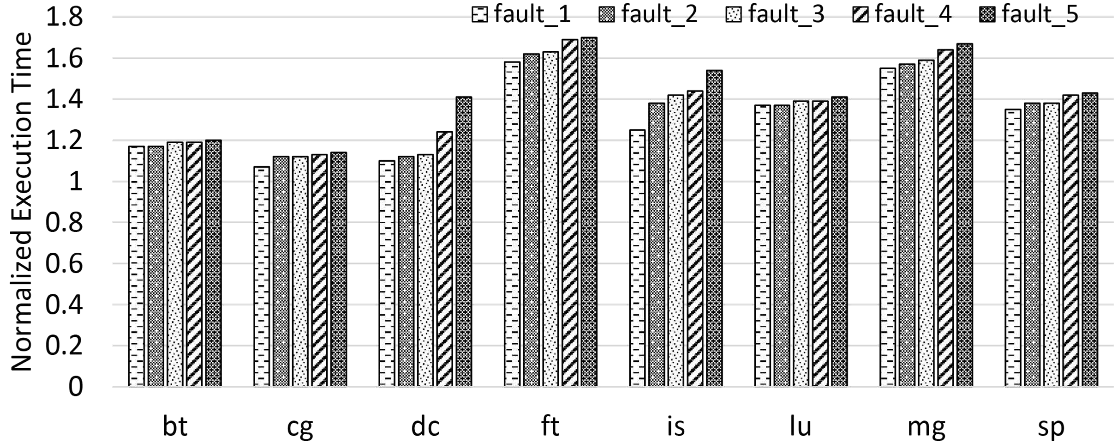

Figure 5.14: Normalized execution time under *Rec_Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates.

Ckpt_F). On average the performance overhead reduction ranges from ≈9% up to 12% for different fault rates under *Rec_Ckpt_F*.

Similar to performance overhead, the EDP also increases when more fault occurs in the execution. Figure 5.15 shows the normalized EDP (w.r.t. *No_Ckpt*) of benchmarks when having varying fault rates under *Ckpt_F* configuration. The general trend is similar to performance overhead, but more exacerbate for EDP.

Under *Rec_Ckpt_F* configuration, the EDP improves up to 48.07% (for is) for single fault, 47.77% (for dc) for two faults, 50.04% (for dc) for three faults, 42.99% (for dc) for four faults, 34.99% (for is) four five faults occur in execution. On average EDP improvement ranges from ≈18% up to 24% for different fault rates under *Rec_Ckpt_F*.

5.6.6 Impact of Checkpoint Frequency on Checkpointing Overhead

The associated overhead of checkpointing is a function of how frequent a checkpoint is established, as well as the amount of states being updated after the most recent checkpoint. Performance and energy overhead of checkpointing increase as the checkpointing frequency increases.

In this section, we aim to analyze the impact of checkpointing frequency on associated checkpointing overhead, and how data recomputation reacts to varying checkpointing frequencies. To do so, we vary checkpoint frequency that yields certain number

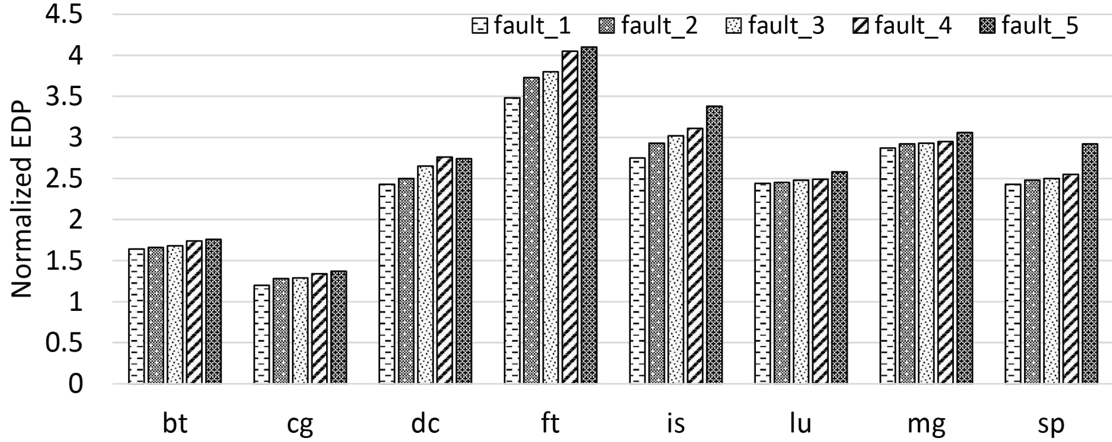

Figure 5.15: Normalized EDP under *Ckpt_F* (w.r.t. *No_Ckpt*) with different fault rates.

of checkpointing interval for each benchmark. We set the checkpointing frequency for benchmarks to have 25, 50, 75 and 100 checkpoint intervals. These checkpoint intervals are uniformly distributed over the execution of the benchmarks.

Figure 5.16 shows the normalized execution time of the benchmarks under *Ckpt_NF* configuration when different checkpoint frequencies are used. In Figure 5.16, ckpt_25 represents the checkpoint frequency that yields to have 25 checkpoint intervals for a given benchmark. Similarly, ckpt_50, ckpt_75, and ckpt_100 represent the checkpointing frequencies that yield to have 50, 75 and 100 checkpoint intervals, respectively. The normalization base is *No_Ckpt*.

Naturally, the performance overhead of checkpointing increases as the checkpoint frequency increases. Among the benchmarks, ft experiences the largest performance overhead under *Ckpt_NF* configuration.

Figure 5.17 shows the normalized execution time of benchmarks under *Rec_Ckpt_NF* configuration when different checkpoint frequencies are employed. General trend is very similar to *Ckpt_NF* configuration; however, *Rec_Ckpt_NF* considerably reduces the performance overhead of checkpointing. An interesting point in Figure 5.17 is the normalized execution time of ckpt_75 is lower than ckpt_50 for is. Although it seems unintuitive at the first place, there is catch in this case. Notice that when we chance checkpointing frequency, the start time of each checkpoint interval becomes different (since we uniformly distribute the checkpoint intervals). The ability of data recomputation to

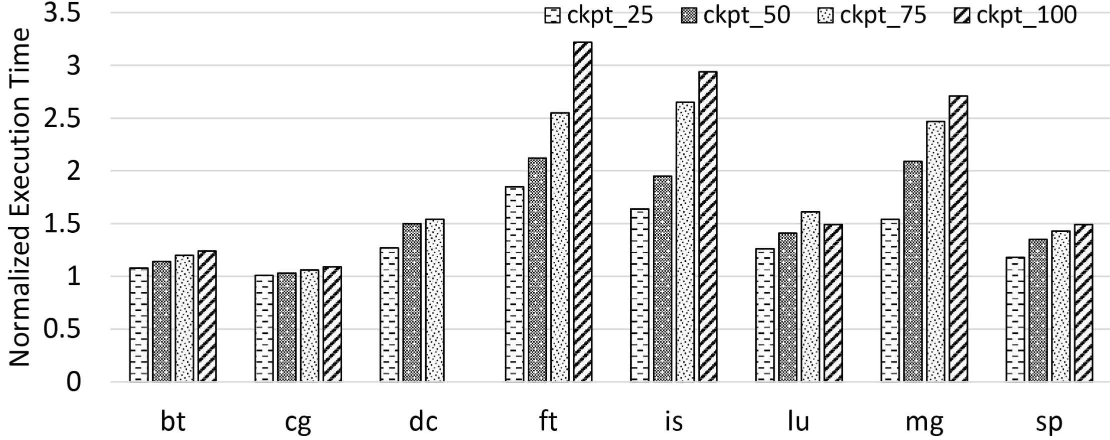

Figure 5.16: Normalized execution time under *Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies.

reduce the checkpoint size (and checkpoint overhead) depends on whether there exist any *RSlice* for that checkpoint interval. If the checkpoints fall into the intervals of the execution where the amount of data that can be recomputed (thus can be excluded from checkpointing) is small, then the benefits of data recomputation can be limited. Such a corner case occurred in is when we run it with checkpoint frequency that yields 50 checkpoints (i.e. ckpt_50). Compared to ckpt_75, the checkpoint intervals under ckpt_50 has limited *RSlice* coverage, meaning the amount of data to be recomputed (i.e. can be excluded from checkpointing) is smaller. So, *Rec_Ckpt_NF* with ckpt_50 has higher performance overhead compared to ckpt_75. The performance overhead of *Ckpt_NF* is reduced up to 28.81% (for is) for ckpt_25, 25.3% (for dc) for ckpt_50, 50.86% (for is) for ckpt_75, and 43.52% (for is) for ckpt_100 under *Rec_Ckpt_NF* (w.r.t. *Ckpt_NF*). On average the performance overhead reduction ranges from ≈10% up to 14% for different checkpoint frequencies under *Rec_Ckpt_NF*.

The similar trend exists for EDP. Figure 5.18 shows the normalized EDP under *Ckpt_NF* configuration for different checkpoint frequencies. On the other hand, *Rec_Ckpt_NF* improves the EDP up to 47.98% (for is) for ckpt_25, 47.74% (for dc) for ckpt_50, 74.19% (for is) for ckpt_75, and 63.45% (for is) for ckpt_100 (w.r.t. *Ckpt_NF*). On average EDP improvement ranges from ≈20% up to 26% for different checkpoint frequencies under *Rec_Ckpt_NF*.

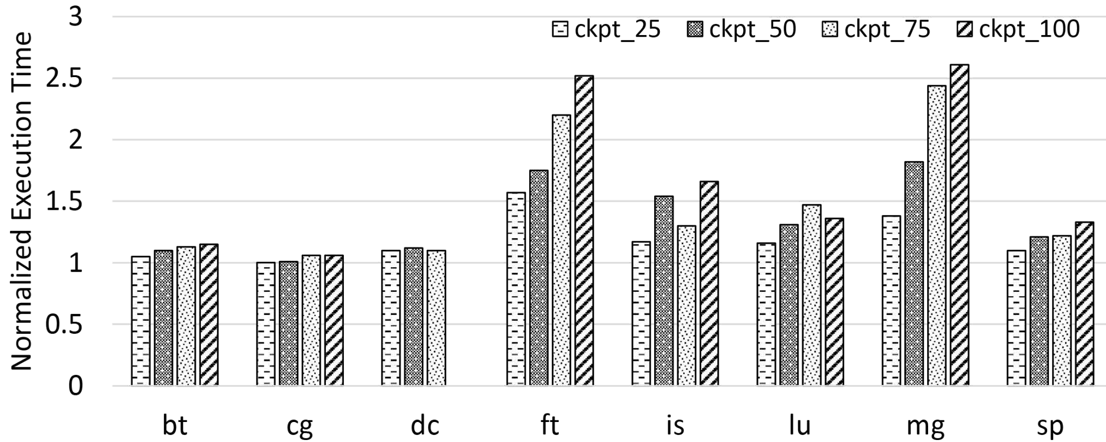

Figure 5.17: Normalized execution time under *Rec_Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies.

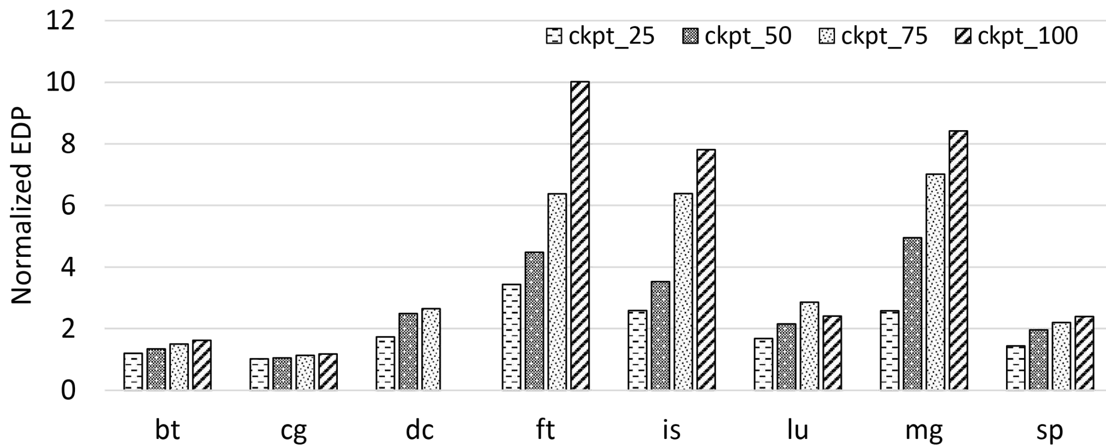

Figure 5.18: Normalized EDP under *Ckpt_NF* (w.r.t. *No_Ckpt*) with different checkpoint frequencies.

5.6.7 Coordinated Local vs. Global Checkpointing

In our discussions and evaluations so far, we focused on global checkpointing since it is simple to implement and easy to understand. It is widely used in practice as well due to its simplicity, so it is a representative option. An alternative to global checkpointing is known as coordinated local checkpointing [60, 51]. Main difference of coordinated local checkpointing is that it does not forces all threads to participate in checkpointing. It is necessary to checkpoint and rollback (in case of fault) threads together that have

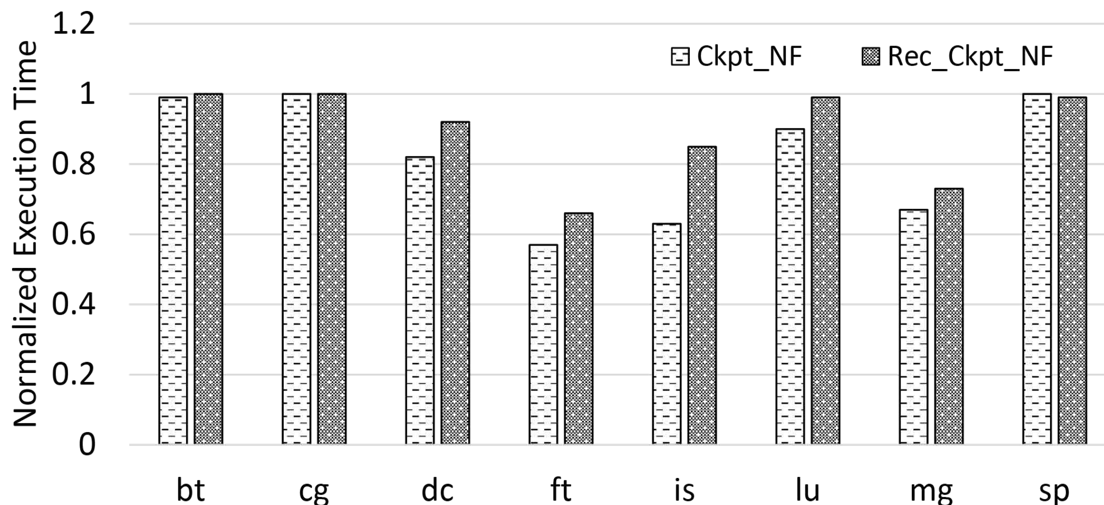

Figure 5.19: Normalized execution time of *Ckpt_NF* and *Rec_Ckpt_NF* for coordinated local checkpointing (w.r.t. *Ckpt_NF* and *Rec_Ckpt_NF* for global checkpointing, respectively).

been communicating for a given checkpoint interval. The other threads that do not participate in communicating may not need to checkpoint at the time others checkpoint. Coordinated local checkpointing is advocated to be scalable, due to associated overheads of checkpoint and recovery is a function of the number of threads that communicate with each other. To identify the threads that communicated with each other within a given checkpoint interval, there has to be a mechanism to track inter-thread data dependencies. Although coordinated checkpointing has an advantage of having reduced set of threads need to checkpoint together, the disadvantage is that identifying communicating threads needs continuous and dynamic monitoring and recording that may not be a challenge for scaling as well.

Without loss of generality, coordinated local checkpointing is, yet another, design point, and in this section we want to evaluate the effectiveness of recomputation-enabled checkpointing for coordinated local checkpointing.

To make a comparison between global and coordinated local checkpointing, we use the *Ckpt_NF* and *Ckpt_F* in global checkpointing as normalization points for *Ckpt_NF* and *Rec_Ckpt_NF* in coordinated local checkpoint, respectively. Similarly, we use the *Ckpt_F* and *Rec_Ckpt_F* in global checkpoint as normalization points for *Ckpt_F* and

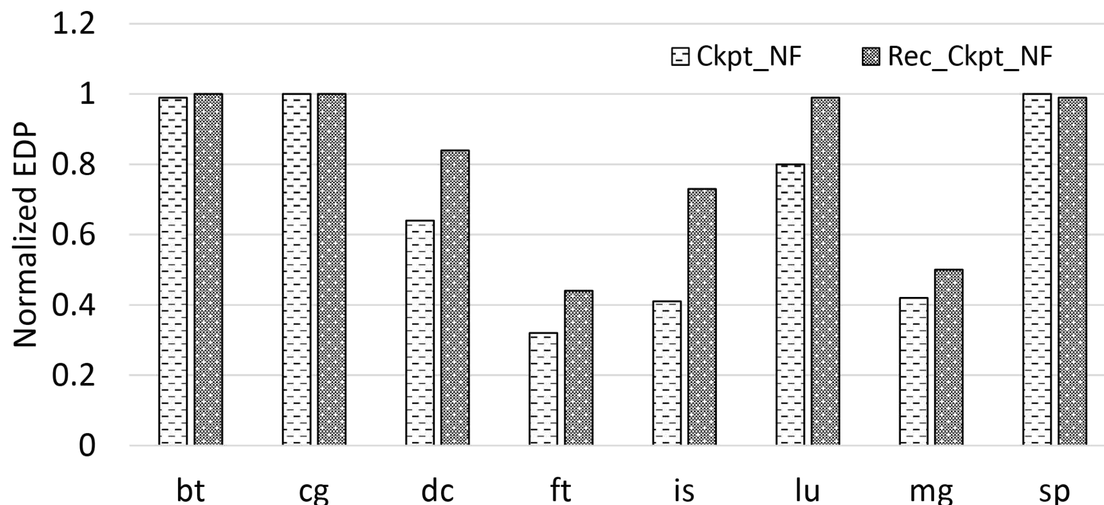

Figure 5.20: Normalized EDP of *Ckpt_NF* and *Rec_Ckpt_NF* for coordinated local checkpointing (w.r.t. *Ckpt_NF* and *Rec_Ckpt_NF* for global checkpointing, respectively).

Rec_Ckpt_F in coordinated local checkpoint, respectively.

Figure 5.19 shows the normalized execution time of benchmarks under *Ckpt_NF* and *Rec_Ckpt_NF* configurations when coordinated local checkpointing is used. As we can see, coordinated local checkpointing reduces the overhead of *Ckpt_NF* in global checkpointing for majority of the benchmarks. The reduction of the overhead is due to the shrinkage of number of threads checkpointing together (i.e. excluding non-communicating threads from checkpointing for a given checkpoint interval). However, there are benchmarks, including bt, cg and sp, that have not seen any overhead reduction under *Ckpt_NF* in coordinated local checkpointing. This is because mainly all the threads are communicating within a given checkpointing interval, so the number of threads involving in checkpointing remains the same in comparison to global checkpointing. For these benchmarks, we do not observe any sizable reduction in performance overhead under *Ckpt_NF* in coordinated local checkpointing. For the rest of the benchmarks the performance overhead of *Ckpt_NF* in coordinated local checkpointing reduces up to ≈42% for ft, 17% for dc, 36% for is, 32% for mg, and 10% for lu (w.r.t. *Ckpt_NF* in global checkpointing).

The recomputation-enabled checkpointing in coordinated local checkpointing remains as effective as it is in global checkpointing. For all the benchmarks, the checkpointing overhead introduced by *Rec_Ckpt_NF* in coordinated local checkpointing remains below (or at most the same) the overhead of *Rec_Ckpt_NF* in global checkpointing. The reductions we observe for *Rec_Ckpt_NF* are not pronounced as much as *Ckpt_NF* in coordinated local checkpointing, mainly because the potential for data recomputation does not change drastically. On the other hand, generally the number of values that can not be recomputed (so have to be checkpointed) reduces more than the ones that can be recomputed. For this reason, *Ckpt_NF* results relatively higher reduction in performance overhead compared to *Rec_Ckpt_NF* in coordinated local checkpointing with respect to their global checkpointing counterparts (i.e. *Ckpt_NF* and *Rec_Ckpt_NF* in global checkpointing, respectively). Among the benchmarks, bt, cg, lu, and sp do not observe any sizable reduction ($\approx\leq 1\%$) on performance overhead of *Rec_Ckpt_NF* in coordinated local checkpointing. For the rest of the benchmarks the performance overhead of *Rec_Ckpt_NF* in coordinated local checkpointing reduces up to ≈8% for dc, 33% for ft, 15% for is, and 26% for mg (w.r.t. *Rec_Ckpt_NF* in global checkpointing).

We observe similar trends for EDP for coordinated local checkpointing. Figure 5.20 shows the normalized EDP of benchmarks under *Ckpt_NF* and *Rec_Ckpt_NF* configurations in coordinated local checkpointing. Compared global checkpointing, EDP reduces under *Ckpt_NF* in coordinated local checkpointing up to 35.68% for dc, 67.15% for ft, 58.26% for is, 19.99% for lu, and 57.92% for mg (w.r.t. *Ckpt_NF* in global checkpointing). On the other hand, EDP reduces under *Rec_Ckpt_NF* in coordinated local checkpointing up to 15.85% for dc, 55.68% for ft, 26.24% for is, and 49.75% for mg (w.r.t. *Rec_Ckpt_NF* in global checkpointing).

Figure 5.21 shows the normalized execution time of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations when coordinated local checkpointing is used. In case of a fault occurs in execution, a rollback and recovery have to be performed. The trends are similar to *Ckpt_NF* and *Rec_Ckpt_NF* configurations in coordinated local checkpointing. One difference is the gap between the execution time of benchmarks performing global checkpointing and coordinated checkpointing gets shrunk. We do not observe any sizable reduction in performance overhead of benchmarks bt, cg, lu and sp under *Ckpt_F* in coordinated local checkpointing. For the rest of the benchmarks the performance

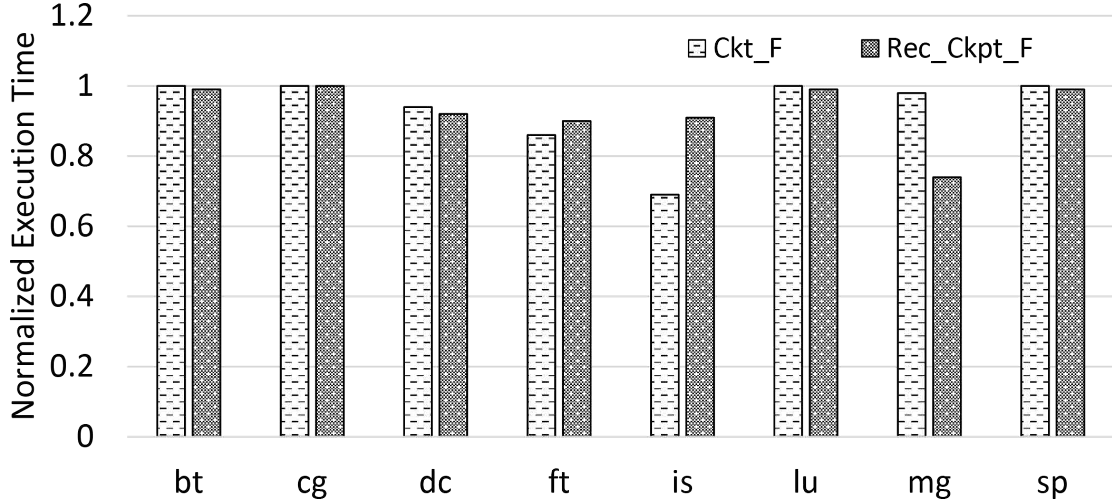

Figure 5.21: Normalized execution time of *Ckpt_F* and *Rec_Ckpt_F* for coordinated local checkpointing (w.r.t. *Ckpt_F* and *Rec_Ckpt_F* for global checkpointing, respectively).

overhead of *Ckpt_F* in coordinated local checkpointing reduces up to ≈14% for ft, 6% for dc, 31% for is, and 2% for mg (w.r.t. *Ckpt_F* in global checkpointing). On the other hand, the performance overhead of *Rec_Ckpt_F* in coordinated local checkpointing reduces up to ≈8% for dc, 10% for ft, 9% for is, and 26% for mg (w.r.t. *Rec_Ckpt_F* in global checkpointing).

Figure 5.20 shows the normalized EDP of benchmarks under *Ckpt_F* and *Rec_Ckpt_F* configurations in coordinated local checkpointing. Compared global checkpointing, EDP reduces under *Ckpt_F* in coordinated local checkpointing up to 18.33% for dc, 33.24% for ft, 51.46% for is, and 11.29% for mg (w.r.t. *Ckpt_F* in global checkpointing). On the other hand, EDP reduces under *Rec_Ckpt_F* in coordinated local checkpointing up to 15.80% for dc, 23.81% for ft, 17.99% for is, and 47.32% for mg (w.r.t. *Rec_Ckpt_F* in global checkpointing).

Based on the outcomes of the evaluations in this section, we can conclude that recomputation-enabled checkpointing and recovery in coordinated local checkpointing is as effective as in global checkpointing (if not more effective).

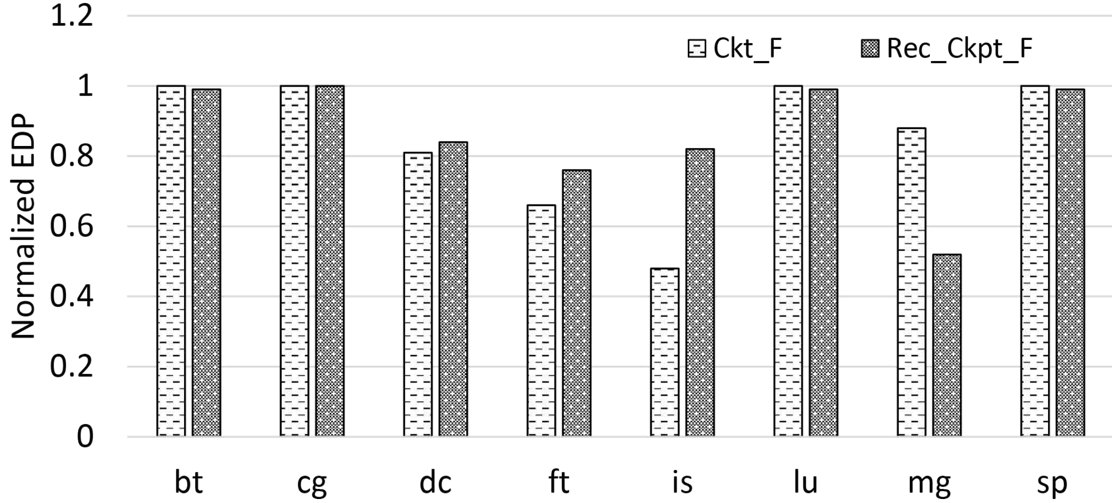

Figure 5.22: Normalized EDP of *Ckpt_F* and *Rec_Ckpt_F* for coordinated local checkpointing (w.r.t. *Ckpt_F* and *Rec_Ckpt_F* for global checkpointing, respectively).

5.6.8 Impact of *RSlice* Length on Checkpoint Size

RSlice length imposes the cost of recomputation. Longer *RSlice*s means higher recomputation cost. In a fault-free execution, the cost of recomputation may be irrelevant, since recomputation is necessary only when there is a fault and recovery is needed. However, in practice, we have to make sure that the execution can resume after detecting a fault and recovering from it, in a low-cost fashion. So, we can not generate recomputation-enabled binary without considering the recomputation of *RSlice*s. For our evaluations, we use a threshold of 10 instructions (except is, where threshold is 5) to identify the *RSlice*s to be embedded into binary. Notice that if we have a higher threshold, there may be more *RSlice*s to be included in binary, so the likelihood of having a value that has a corresponding *RSlice* increases. This means that the number of values that can be recomputed (thus can be eliminated from checkpoint) may increases. As a result the checkpoint size gets reduced. As an example, Figure 5.23 shows the impact of *RSlice* length on reduction of total checkpoint size under *Rec_Ckpt_NF* configuration for bt. The data labels on x axis of the Figure 5.23 represents the threshold used in selection of *RSlice*s. The label length_50 means the threshold is 50 (i.e. *RSlice* can have at most 50 instructions), length_40 means the threshold is 40, and so on. The total checkpoint size reduces up to 89.91% when *RSlice* length is allowed to grow up to 50 instructions,

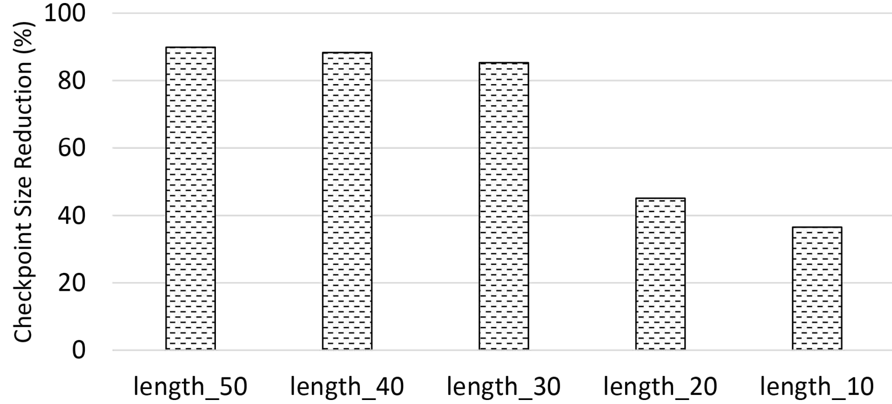

Figure 5.23: Total checkpoint size reduction as a function of *RSlice* length for bt.

and 36.54% when the *RSlice* length remains less than or equal to 10. One should pay a special attention while choosing the threshold. It has the impact on recomputation cost (during recovery in case of fault), and the microarchitectural support needed to facilitate data recomputation. In our evaluations, we pick conservative threshold to keep the microarchitectural resources needed reasonable (as *RSlice*s length increases, we may need bigger Hist table), and not to favor recomputation-enabled checkpointing unfairly.

We expect the values that have corresponding *RSlice*s and can be recomputed are not uniformly distributed among the checkpoint intervals. This means for each checkpoint interval, we may observe varying levels of benefits from data recomputation. That variation translates into variation on checkpoint size reduction over checkpointing intervals. Figure 5.24 shows how the effectiveness of recomputation-enabled checkpointing on reducing checkpoint size changes over time for bt (when using different thresholds for *RSlice* length). We see that *Rec_Ckpt_NF* reduces checkpoint size more for certain checkpoint intervals compared to other intervals. Such kind of variation can be exploited for improving the impact of recomputation-enabled checkpointing. The checkpointing frequency would be changed dynamically to perform the checkpointing when there exist high potential for recomputation (i.e. checkpoint intervals where the number of values that can be recomputed is high). We do not investigate on such kind of dynamic and intelligent scheme in the scope of this dissertation, rather we just want to motivate for further research on how recomputation-enabled checkpointing can be extended further.

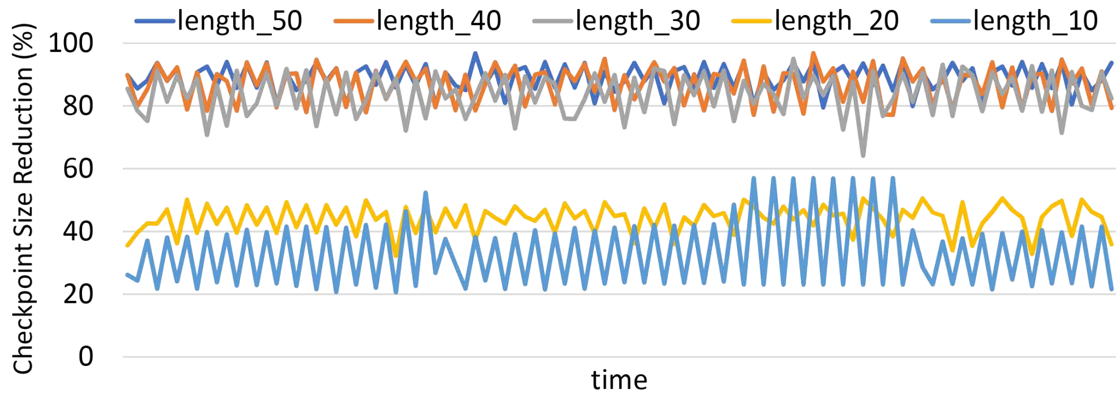

Figure 5.24: Impact of *RSlice* length on checkpoint size over time for bt.

The analysis of the impact of *RSlice*s length for the rest of the benchmarks can be found in Appendix A.

5.7 Related Work

The fault-tolerant system design and checkpointing are extensively studied over the decades. The proposed solutions can be categorized into software-based, hardware-based checkpointing; application or system level checkpointing. Software-based proposals use periodic barriers to perform system-level [61], application-level [62], or hybrid checkpoints [63].

Hardware proposals [56, 55, 51] reduce the checkpoint and restart penalties, but introduce hardware complexity. In Rebound [56] when a core is checkpointing, the L2 controller writes dirty lines back to main memory while keeping the clean copy in L2. Memory controller logs the old value of the updated memory address. In addition, between checkpoint times, when a dirty cache line is written back to memory, memory controller also logs the old value. This is done for the first writeback and consecutive writes to the same memory address can be eliminated from being logged. SafetyNet [51] explicitly checkpoints register file, and incrementally checkpoints the memory state by logging the previous value.

Compiler-assisted checkpointing [64] improves the performance of automated checkpointing by presenting a compiler analysis for incremental checkpointing, aiming to reduce checkpoint size. In incremental checkpointing, the memory updates are monitored

and the updates are omitted from checkpointing if it is detected a particular memory location has not been modified between two adjacent checkpoint. This mechanism reduces the amount of data to be checkpointed and widely used in many checkpointing schemes. We also employ incremental checkpointing in our analysis. In [64], instead of using runtime mechanisms (such as exploiting cache coherency protocol to identify the updates memory locations), they rely on compiler analysis to track the memory updates that can be excluded from checkpoint. To facilitate the compiler analysis, the source code should be manually annotated, indicating the starting point of the checkpoint. However, it has limited applicability in practice, since it may not be always feasible to obtain the source code.

A relevant work presented in [39], introduces the notion of idempotent execution and corresponding architecture that does not require to have explicit checkpoints to recover from misspeculation or fault. In case of misspeculation or fault it is only necessary to re-execute the idempotent region to recover. Such idempotent regions are constructed by the compiler. As the name suggests, idempotent regions regenerates the same output regardless of how many times it is executed with the given program state. In comparison to our recomputation-enabled checkpointing and recovery, idempotent execution has limited flexibility. Generally, idempotent regions are large, meaning they incur high overhead during recovery, while we employ fine-grained data recomputation (separate *RSlice* for each value), and each *RSlice* contains only necessary instructions which generally tends be limited in number. Generating idempotent regions is also daunting task. It may not be easy to find and generate fine-grained idempotent regions for the large class of applications which limits the effectiveness of idempotent execution for eliminating checkpointing overheads and minimizing recovery overheads. *RSlice*s provide more flexibility on values to be checkpointed and be recomputed, so our recomputation-enabled checkpoint and recovery scheme has wider applicability. The idempotent execution is also explored in the context of recovering from concurrency bugs [65]. In this work, we study how recomputation can help mitigate the performance and energy overhead of checkpointing, as well as rollback and recovery, assuming a microarchitectural support needed for data recomputation.

Dong et al. [66] proposed two-level hybrid local/global checkpointing to reduce the

checkpointing overhead. The main idea is to take frequent local checkpoints for recovering transient failures (where recovery can be achieved by relying on local checkpoint) and less frequent global checkpoint for recovering failures that require global recovery. Since local checkpoints incur less overhead, reducing the number of global checkpoints leads to overall reduction on checkpointing overhead. Furthermore, they used non-volatile memory (PCRAM) as a checkpointing medium to provide no-leakage, high bandwidth and fast access memory.

Chapter 6

Conclusion

Technology scaling and innovative architecture-level solutions to date have improved the energy efficiency of data generation, i.e., computation, significantly more than the energy efficiency of data communication [3, 4]. As a result, both, time and power spent in communication highly exceed the time and power spent in computation. As a consequence, recomputing data can become more energy-efficient than storing and retrieving pre-computed data.

In Chapter 3, we investigate the effectiveness of recomputing data values in minimizing, the overhead of expensive off-chip memory accesses. The idea is replacing a load with a sequence of instructions to recompute the respective data value, only if it is more energy-efficient. We call the resulting execution model *amnesic*. We detail an illustrative proof-of-concept design, identify practical limitations, and provide design guidelines. The proof-of-concept implementation features an amnesic compiler, microarchitectural support for amnesic execution, and an instruction scheduler to orchestrate amnesic execution at runtime. Overall, we find that amnesic execution can reduce energy-delay-product of sequential execution by up to 87%, 24.92% on average, for 11 out of 33 benchmarks deployed.

In Chapter 4, we explore (interactions between) two broad classes of recomputation techniques: brute-force recalculation and prediction based recomputation. Under recalculation, the recomputation effort goes to the generation of the data values (which would otherwise be loaded from memory), by re-executing the producer instruction(s)

of the eliminated load(s). Under prediction, the recomputation effort goes to the estimation of the data values by exploiting value locality – the likelihood of the recurrence of values (which would otherwise be loaded from memory) within the course of execution. We find that recalculation has wider coverage for recomputation than prediction, as prediction cannot be effective under limited value locality.

In the presence of errors, periodic checkpointing of the machine state makes recovery of execution from a safe state possible. The performance and energy overhead of both checkpointing and recovery, however, can get overwhelming with the frequency of checkpointing and anticipated errors. In Chapter 5 we discuss how recomputation of data values can help mitigate such overheads and quantitatively characterize recomputation-enabled checkpointing. We observe that recomputation can reduce the total checkpoint size by 38.31%, and memory footprint of checkpoints by 23.91%. Similarly, the performance, energy and EDP overhead can be reduced by 11.92%, 12.53%, and 23.41%, respectively.

References

[1] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *International Symposium on Computer Architecture*, 2011.

[2] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference*, 2012.

[3] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Lucas Robert, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA Information Processing Techniques Office (IPTO) sponsored study*, 2008.

[4] Mark Horowitz. Computing's Energy Problem (and what we can do about it). *Keynote at International Conference on Solid State Circuits*, April 2014.

[5] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5), 1988.

[6] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5), 2011.

[7] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[8] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *International Symposium on Computer Architecture*, 2002.

[9] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of Load/Store Instruction Latencies. In *International Symposium on Microarchitecture*, 1993.

[10] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4), 2006.

[11] David H. Bailey, Eric Barszcz, J. Tyler Barton, David S. Browning, Ronald L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, Horst D. Simon, Venkatesan Venkatakrishnan, and Sisira K. Weeratunga. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Conference on High Performance Computing Networking, Storage and Analysis*, 1991.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, 2005.

[15] Yakun S. Shao and David Brooks. Energy characterization and instruction-level energy model of intel's Xeon Phi processor. In *International Symposium on Low-Power Electronics and Design*, pages 389–394, September 2013.

[16] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture*, December 2009.

[17] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2011.

[18] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, 1996.

[19] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load Value Approximation. In *International Symposium on Microarchitecture*, 2014.

[21] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, May 2014.

[22] Erin Carson, Nicholas Knight, and James Demmel. Avoiding communication in nonsymmetric lanczos-based krylov subspace methods. *SIAM Journal on Scientific Computing*, 35(5), 2013.

[23] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[24] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[25] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Programming Language Design and Implementation*, 1992.

[26] Mahmut Kandemir, Feihul Li, Guilin Chen, Guangyu Chen, and Ozcan Ozturk. Studying storage-recomputation tradeoffs in memory-constrained embedded processing. In *Design, Automation and Test in Europe*, 2005.

[27] Hakduran Koc, Ozcan Ozturk, Mahmut Kandemir, and Ehat Ercanli. Minimizing Energy Consumption of Banked Memories Using Data Recomputation. In *International Symposium on Low-Power Electronics and Design*, 2006.

[28] Hakduran Koc, Mahmut Kandemir, Ehat Ercanli, and Ozcan Ozturk. Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-processors. In *Design Automation Conference*, 2007.

[29] Doug Burger, Stefanos Kaxiras, and James R. Goodman. Datascalar Architectures. In *International Symposium on Computer Architecture*, 1997.

[30] Harold S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, C-19(1), January 1970.

[31] Peter M. Kogge. The EXECUBE approach to massively parallel processing. 1994.

[32] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *International Symposium on Microarchitecture*, 17(2):34–44, 1997.

[33] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas. Flexram: toward an advanced intelligent memory system. In *International Conference on Computer Design*, 1999.

[34] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *International Symposium on Computer Architecture*, 1998.

[35] Peter M. Kogge, Steven C. Bass, Jay B. Brockman, Danny Z. Chen, and Edwin Sha. Pursuing a petaflop: point designs for 100 TF computers using PIM technologies. In *Frontiers of Massively Parallel Computing*, 1996.

[36] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture*, 1998.

[37] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *International Symposium on Computer Architecture*, 1997.

[38] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing. In *International Symposium on Computer Architecture*, 2010.

[39] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent Processor Architecture. In *International Symposium on Microarchitecture*, 2011.

[40] Craig Zilles and Gurindar Sohi. Execution-based Prediction Using Speculative Slices. In *International Symposium on Computer Architecture*, 2001.

[41] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *International Symposium on Computer Architecture*, 2001.

[42] Amir Roth and Gurindar S. Sohi. A quantitative framework for automated pre-execution thread selection. In *International Symposium on Microarchitecture*, 2002.

[43] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-processors: An Implementation of Operation-based Prediction. In *International Conference on Supercomputing*, 2001.

[44] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[45] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. The Load Slice Core Microarchitecture. In *International Symposium on Computer Architecture*, 2015.

[46] Jian Huang and D. J. Lilja. Exploiting Basic Block Value Locality With Block Reuse. In *International Symposium on High Performance Computer Architecture*, pages 106–114, 1999.

[47] Kwei-Jay Lin, Swaminathan Natarajan, and Jane W.-S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Real-Time Systems Symposium*, 1987.

[48] Ismail Akturk and Ulya R. Karpuzcu. AMNESIAC: Amnesic Automatic Computer - Trading Computation for Communication for Energy Efficiency. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[49] Hui Jin, Yong Chen, Huaiyu Zhu, and Xian-He Sun. Optimizing hpc fault-tolerant environment: An analytical approach. In *International Conference on Parallel Processing*, 2010.

[50] Elmootazbellah N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), September 2002.

[51] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, 2002.

[52] Dwight Sunada, Michael Flynn, and David Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *International Symposium on Fault-Tolerant Computing*, 1999.

[53] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.

[54] Christine Morin, Alain Gefflaut, Michel Banâtre, and Anne-Marie Kermarrec. Coma: An opportunity for building fault-tolerant scalable shared memory multiprocessors. In *International Symposium on Computer Architecture*, 1996.

[55] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, 2002.

[56] Rishi Agarwal, Pranav Garg, and Josep Torrellas. Rebound: Scalable checkpointing for coherent shared memory. In *International Symposium on Computer Architecture*, 2011.

[57] Seong-Lyong Gong, Minsoo Rhu, Jungrae Kim, Jinsuk Chung, and Mattan Erez. Clean-ecc: High reliability ecc for adaptive granularity memory system. In *International Symposium on Microarchitecture*, 2015.

[58] Timothy J. Dell. A white paper on the benefits of chipkill- correct ecc for pc server main memory. 1997.

[59] Shuou Nomura, Matthew D. Sinclair, Chen-Han Ho, Venkatraman Govindaraju, Marc de Kruijf, and Karthikeyan Sankaralingam. Sampling + dmr: Practical and low-overhead permanent fault detection. In *International Symposium on Computer Architecture*, 2011.

[60] Michel Banâtre, Alain Gefflaut, Philippe Joubert, Christine Morin, and Peter A. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, October 1996.

[61] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2005.

[62] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[63] Daniel Marques, Greg Bronevetsky, Rohit Fernandes, Keshav Pingali, and Paul Stodghil. Optimizing checkpoint sizes in the c3 system. In *International Parallel and Distributed Processing Symposium*, 2005.

[64] Greg Bronevetsky, Daniel J. Marques, Keshav K. Pingali, Radu Rugina, and Sally A. McKee. Compiler-enhanced incremental checkpointing for openmp applications. In *Symposium on Principles and Practice of Parallel Programming*, 2008.

[65] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[66] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

Appendix A

Impact of *RSlice* Length on Checkpoint Size

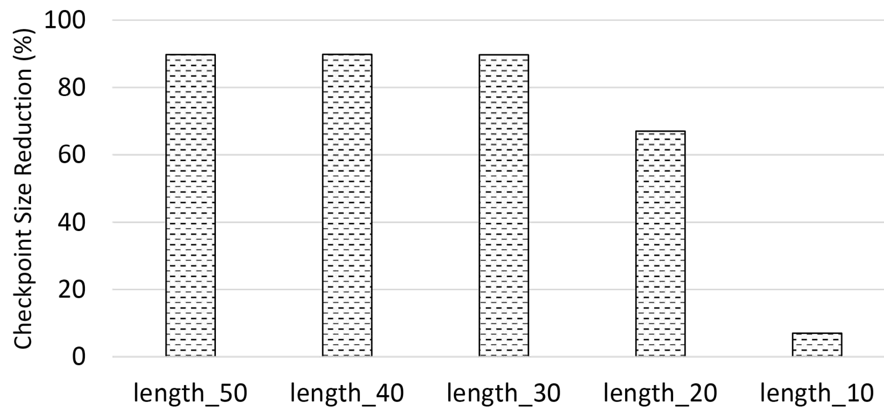

Figure A.1: Total checkpoint size reduction as a function of *RSlice* length for cg.

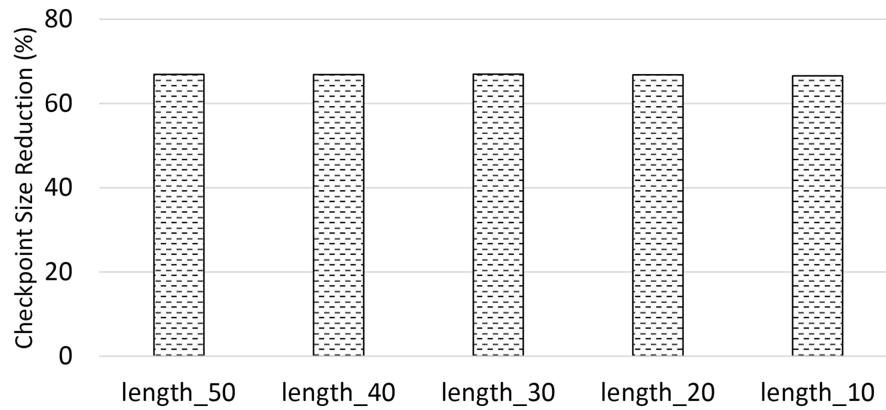

Figure A.2: Total checkpoint size reduction as a function of *RSlice* length for dc.

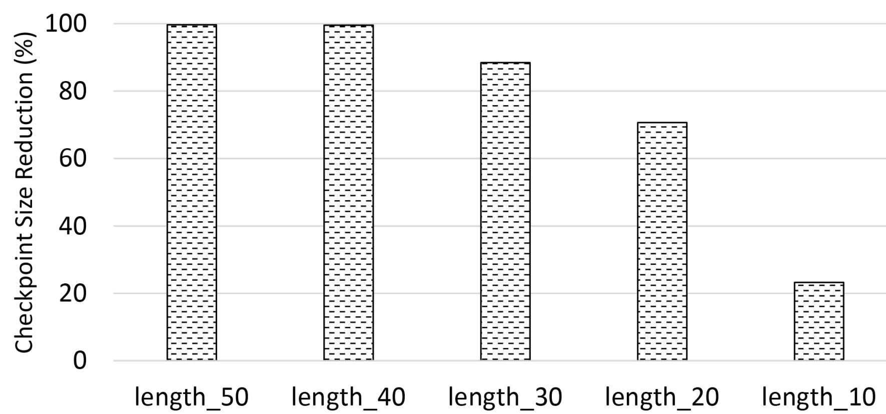

Figure A.3: Total checkpoint size reduction as a function of *RSlice* length for ft.

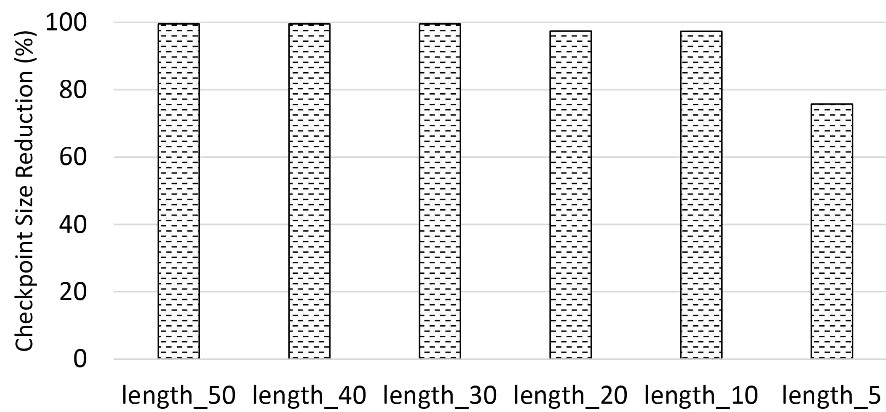

Figure A.4: Total checkpoint size reduction as a function of *RSlice* length for is.

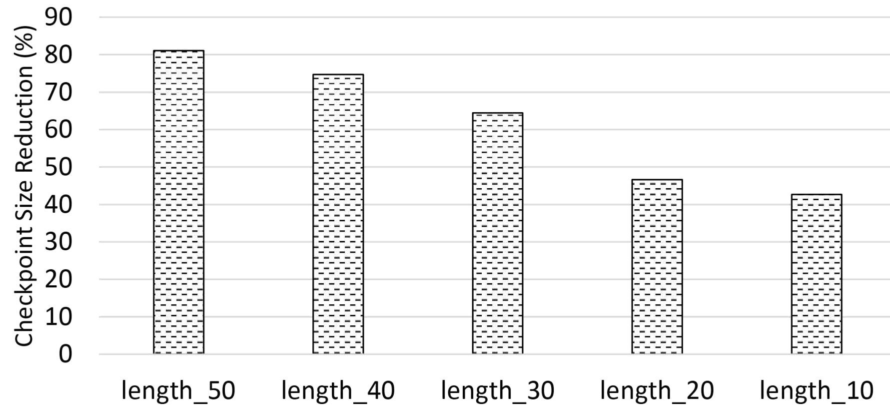

Figure A.5: Total checkpoint size reduction as a function of *RSlice* length for lu.

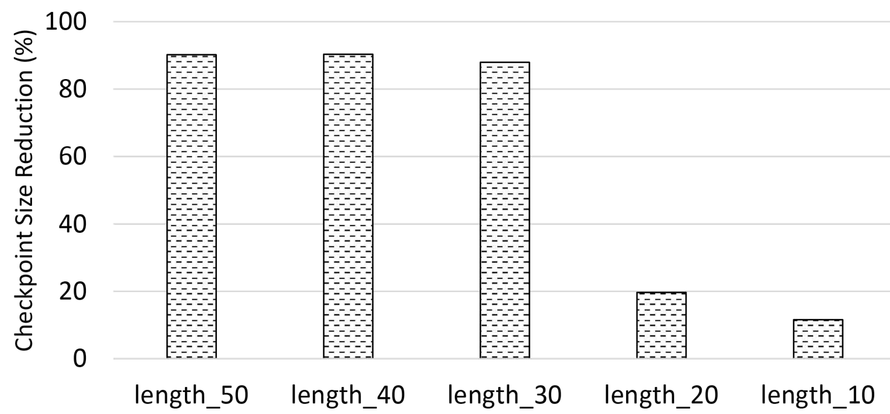

Figure A.6: Total checkpoint size reduction as a function of *RSlice* length for mg.

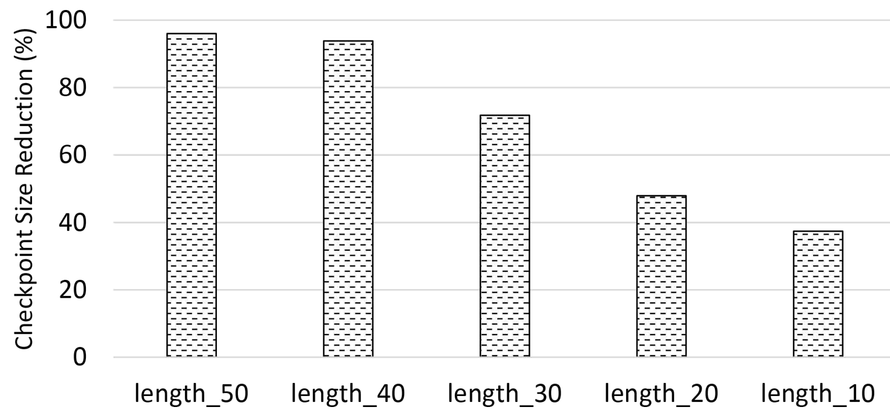

Figure A.7: Total checkpoint size reduction as a function of *RSlice* length for sp.