Abstract

In designing a new processor, computer architects consider a myriad of possible organizations and designs to decide which best meets the constraints on performance, power and cost for each particular processor. To identify practical designs, architects need to have insight into the physical-level characteristics (delay, power and area) of various components of modern processors implemented in recent fabrication technologies. During early stages of design exploration, however, developing physical-level implementations for various design options (often in the order of thousands) is impractical or undesirable due to time and/or cost constraints. In lieu of actual measurements, analytical and/or empirical models can offer reasonable estimates of these physical-level characteristics. However, existing models tend to be out-dated for three reasons: *(i)* They have been developed based on old circuits in old fabrication technologies; *(ii)* The high-level designs of the components have evolved and older designs may no longer be representative; and, *(iii)* The overall architecture of processors has changed significantly, and new components for which no models exist have been introduced or are being considered.

This thesis studies three key components of modern high-performance processors: *Counting Bloom Filters* (CBFs), *Checkpointed Register Alias Tables* (RATs), and *Compacted Matrix Schedulers* (CMSs). CBFs optimize membership tests (e.g., whether a block is cached). RAT and CMS increase the opportunities for exploiting instruction-level parallelism; RAT is the core of the renaming stage, and CMS is an implementation for the instruction scheduler. Physical-level studies or models for these components have been limited or non-existent. In addition to investigating these components at the physical level, this thesis *(i)* proposes a novel speed- and energy-efficient CBF implementation; *(ii)* studies how the number of RAT checkpoints affects its latency and energy, and overall processor performance; and, *(iii)* studies the CMS and its accompanying logic at the physical level. This thesis also develops empirical and analytical latency and energy models that can be adapted for newer fabrication technologies. Additionally, this thesis proposes physical-level latency and energy optimizations for these components motivated by design inefficiencies exposed during the physical-level study phase.

Table of Contents

Table of Contents

Table of Contents

List of Figures

List of Figures

List of Abbreviations

CAM	*Content Addressable Memory*
CBF	*Counting Bloom Filters*
CMS	*Compacted Matrix Scheduler*
cRAT	*CAM-Based Register Alias Table*
DRAM	*Dynamic Random Addressable Memories*
GC	*Global Checkpoint*
ILP	*Instruction Level Parallelism*
IPC	*Instruction Per Cycle*
IW	*Issue Width*
L-CBF	*LFSR-Based Counting Bloom Filters*
LFSR	*Linear Feedback Shift Register*
LSQ	*Load-Store Queues*
NoE	*Number of Entries*
NoGC	*Number of Global Checkpoint*
NoRP	*Number of Rows per Partition*
RAB	*Random Access Buffer*
RAT	*Register Alias Table*
ROB	*Re-Order Buffer*
SAB	*Serial Access Buffer*
S-CBF	*SRAM-Based Counting Bloom Filters*
SRAM	*Static Random Addressable Memory*
sRAT	*SRAM-Based Register Alias Table*
TLB	*Translation Look-aside Buffer*
WAT	*Wakeup Allocation Table*
WoE	*Width of Entries*
WS	*Window Size*

1. Introduction

Designing and implementing a processor requires expertise in several areas including computer architecture and circuit design. Computer architects consider various design options to decide which best meets the performance, power and cost constraints for each particular processor. Although other possibilities exist, the typical goal in designing a new processor is maximizing performance while taking cost and power constraints into consideration. To achieve this goal, computer architects must take into account both software (applications) and fabrication technology constraints and needs (Figure 1-1). Architects are primarily concerned with the functionality and interaction of the processor components. However, architects also need to have insight into the physical-level characteristics of various design options evaluated early in the design process to choose a design compliant with performance, power and cost constraints.

Typically, cost directly relates to area which in turn is limited by different technology considerations. Thus, architects must typically choose designs that fit within given area constraints. Another cost consideration is complexity; developing a complex component may take a long time, making it more prone to design errors and ultimately resulting in a lower performance than a simpler alternative. In addition to area cost and complexity, architects must also consider power constraints to limit heat generated by various processor components as a byproduct. This heat must be safely and economically dissipated not to compromise the operation and integrity of the processor.

In designing a new processor, computer architects have to consider various combinations of the components with different functionality and characteristics. For these components, architects

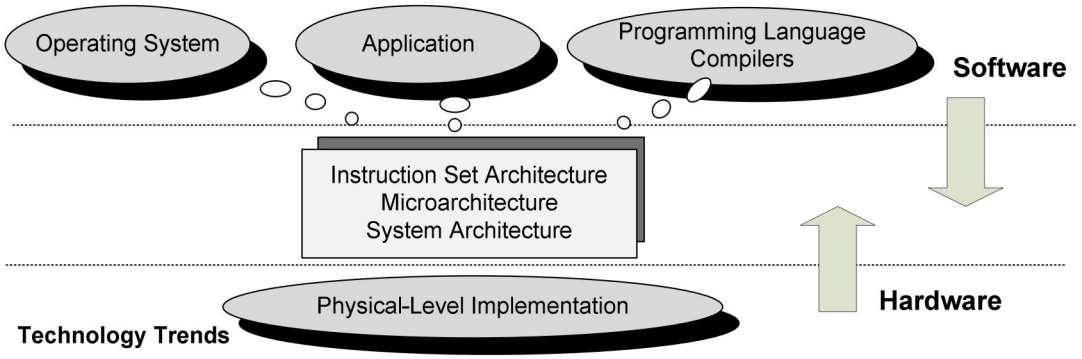


Figure 1-1: Computer architecture and its interaction with software and semiconductor technology.

need to know the expected area, power and performance of each design option with reasonable accuracy. Architects evaluate the performance of processor designs by simulating the cycle-level behavior of the processor. Typically, tens of thousands of simulations are required to evaluate many different processor configurations running a variety of benchmarks, each with multiple different input parameters. A typical simulation run models the cycle-by-cycle behavior of the processor while executing several hundreds and often several billion of instructions. A simulator is a software program that models this cycle-by-cycle processor behavior. Modeling just the execution of a single instruction requires at least several tens of thousands of instructions in reasonably-detailed simulation. As a result, these simulation-based experiments are time consuming requiring several years of simulation time even for relatively simple processor designs. The performance evaluation results of these simulations are typically in terms of the number of cycles various designs would require to execute a particular workload. The actual performance (execution time), however, also depends on the actual cycle length (clock period), which is determined by the physical-level implementation. Hence, architects require latency estimates of various processor components as well as area and power estimates to prune impractical designs. Because the number of candidate designs is very large (e.g., in the order of thousands if not more), estimating latency, energy and area through a physical-level implementation is undesirable or impractical due to time and/or cost constraints. Hence, a different methodology is needed to reasonably estimate these physical-level characteristics, representing the hardware complexity of a design. Models can provide these early estimates to help prune the design search space by identifying impractical designs.

This thesis develops such models for three components of modern high-performance processors: *(i) Counting Bloom Filters* (CBFs), *(ii) Checkpointed Register Alias Tables* (RATs), and *(iii) Compacted Matrix Schedulers* (CMSs). The last two components are specific to dynamically-scheduled processors (addressed in Section 1.2). However, CBFs can be used for other processors as well, and their analysis results are independent of the actual processor design.

The rest of this chapter is organized as follows. Section 1.1 explains the need for modeling the latency and energy of the various processor components and highlights the objectives of this thesis. The functionality and characteristics of the components studied in this thesis are

thoroughly discussed in their respective chapters; however, their role in the datapaths of modern processors is reviewed in Section 1.2. Specifically, Section 1.2 reviews the key components of modern, dynamically-scheduled superscalar processors— their organization is representative of many other modern processors. Section 1.3 briefly introduces the components studied in this thesis, and Section 1.4 presents the organization details of the whole thesis.

1.1 Problem Statement and Objectives

Figure 1-2 shows the typical modeling and design flow for developing a new processor [13]-[15]. The architecture definition for a processor is an iterative process whose result is a physical-level realization at the end of the design flow. At the very early stage of the design process, computer architects evaluate various design options to find the best possible design in accordance with the specification, which defines a desirable level of performance under speed, power and area constraints. The purpose of this architectural-level design exploration is to decide upon the best microarchitectural parameters for a design that will be reasonably compliant with the specification when the design is implemented at the physical level.

As a design is refined from the architectural level to the physical level, the accuracy and details of its functionality, timing, power consumption and area information increase. Specifically, accurate measurements of the aforementioned parameters, representing the hardware complexity characteristics of a design, will be available at the end of the design flow after physical-level implementation. However, estimations of the aforementioned parameters for all explored design options are required at architectural-level as developing physical-level implementations at this stage is impossible or unaffordable due to the time and/or cost constraints.

Analytical and empirical models can provide these early estimations. Models characterize delay, power and area of a design option using a similar-enough physical-level implementation [14]. Although such models exist, developing new models is still necessary because existing publicly-available models have several limitations. First, most publicly-available models are based on

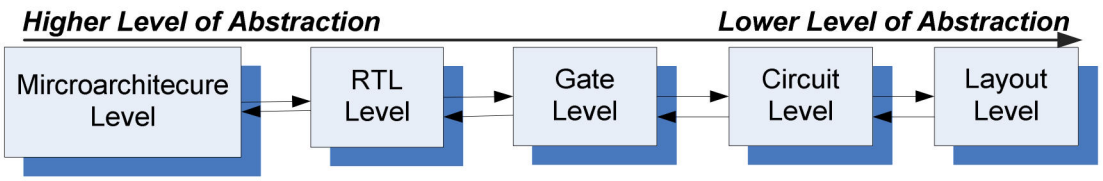


Figure 1-2: Typical flow in a processor design process.

out-dated circuits developed in older fabrication technologies. For instance, consider CACTI, a tool providing latency, power and area estimations for caches [68]. CACTI models were originally developed based on circuits implemented in a 0.8 µm fabrication technology. Second, most high-level estimation tools employ ideal linear scaling to adapt their model estimations for smaller feature sizes. However, many aspects of deep sub-micron technology scaling are far from ideal. Third, the circuit structures developed in older technologies may be inappropriate for newer, deep sub-micron technologies because different challenges have emerged (e.g., leakage power and interconnect delay).

In some cases, no analytical or empirical models for certain processor components exist because they have only been recently proposed (e.g., CBFs and CMSs). In other cases, existing models fail to capture all the functionality of a component. For example, for the checkpointed RAT, existing models do not capture the effect of checkpoints. In these cases, developing such models helps evaluate the effectiveness of the proposed techniques or help prove the validity of an optimization at the physical and/or architectural level.

Considering the previously-mentioned shortcomings in the optimization and modeling areas, this thesis has three principal objectives:

- Investigate the *hardware complexity characteristics (speed, power and area)* for several key components of modern high-performance processors.
- Develop latency and energy analytical and/or empirical models for these components.
- Propose architectural- and/or physical-level optimizations for these components.

To study each of the candidate components, we follow a methodology comprising several steps: First, we develop a physical (circuit and layout) level implementation for a component in a reasonably recent fabrication technology. We apply various delay and/or power optimizations to the circuit depending on the optimization objectives. Then, we use analytical and empirical approaches to develop models for estimating the component's delay and power. Through the physical-level implementation and modeling process, we gain insight into the physical-level characteristics of the component. Having both architectural-level and physical-level information and evaluation results for the component, we can *(i)* explore various architectural alternatives, *(ii)* identify inefficiencies of existing designs, and *(iii)* propose alternative speed- and/or energy-efficient implementations.

1.2 Key Components of Modern Processors

This section reviews the key components of modern, dynamically-scheduled superscalar processors — a selection of these components has been the focus of this thesis.

In the simplest execution model, processors *sequentially* execute program instructions in the order specified by the programmer. In this model, each instruction executes atomically, i.e., at any given point of time only one instruction executes, and an instruction can start execution once its immediately preceding instruction finishes executing. However, the same results can still be achieved with a relaxed execution order at runtime. Consider two adjacent instructions, A and B, that do not change the control flow (i.e., once A executes, B will execute too) — the following concepts can be extended to control-flow instructions as well. Sequential execution requires that these two instructions execute in order. However, B could execute in parallel with A, or even before it as long as B does not require a value produced by A and does not change any state (e.g., register or memory values) that A reads or changes (i.e., B is *independent* of A). This concept is called *instruction-level parallelism (ILP),* and implies that instructions can, under some circumstances, execute in parallel even though the program was written assuming it would be executed sequentially. Executing instructions in parallel (in part or completely) increases concurrency and reduces the overall time required to execute a program (or increases the throughput) while it does not necessarily reduce the time required to execute each instruction. Modern processors use the following micro-architectural level techniques to exploit ILP:

(i) ***Instruction pipelining*****:** The execution of multiple instructions is overlapped in part. Even dependent instructions can be pipelined.

(ii) ***Superscalar execution*****:** Multiple, sequential instructions are executed in parallel utilizing multiple functional units.

(iii) ***Out-of-order execution*****:** Instructions are executed out of the program order as long as no dependencies among instructions are violated. Out-of-order execution technique is orthogonal to pipelining and superscalar techniques.

(iv) ***Register renaming*****:** Without register renaming, an instruction B is dependent on a preceding instruction A if B reads a value produced by A (true dependency), or B overwrites a value that A reads (write-after-read dependency) or writes (write-after-write

dependency. Register renaming eliminates the last two dependencies, called *false (*or *name)* dependencies, to help exploit more ILP.

(v) ***Speculative execution*:** This technique supports the execution of instructions before knowing with certainty whether they should be executed. In control-flow speculation, the most common form of speculative execution, the processor executes instructions following a control-flow instruction (e.g., branch) without definitely knowing which direction the control-flow instruction will take. Without speculative execution, the

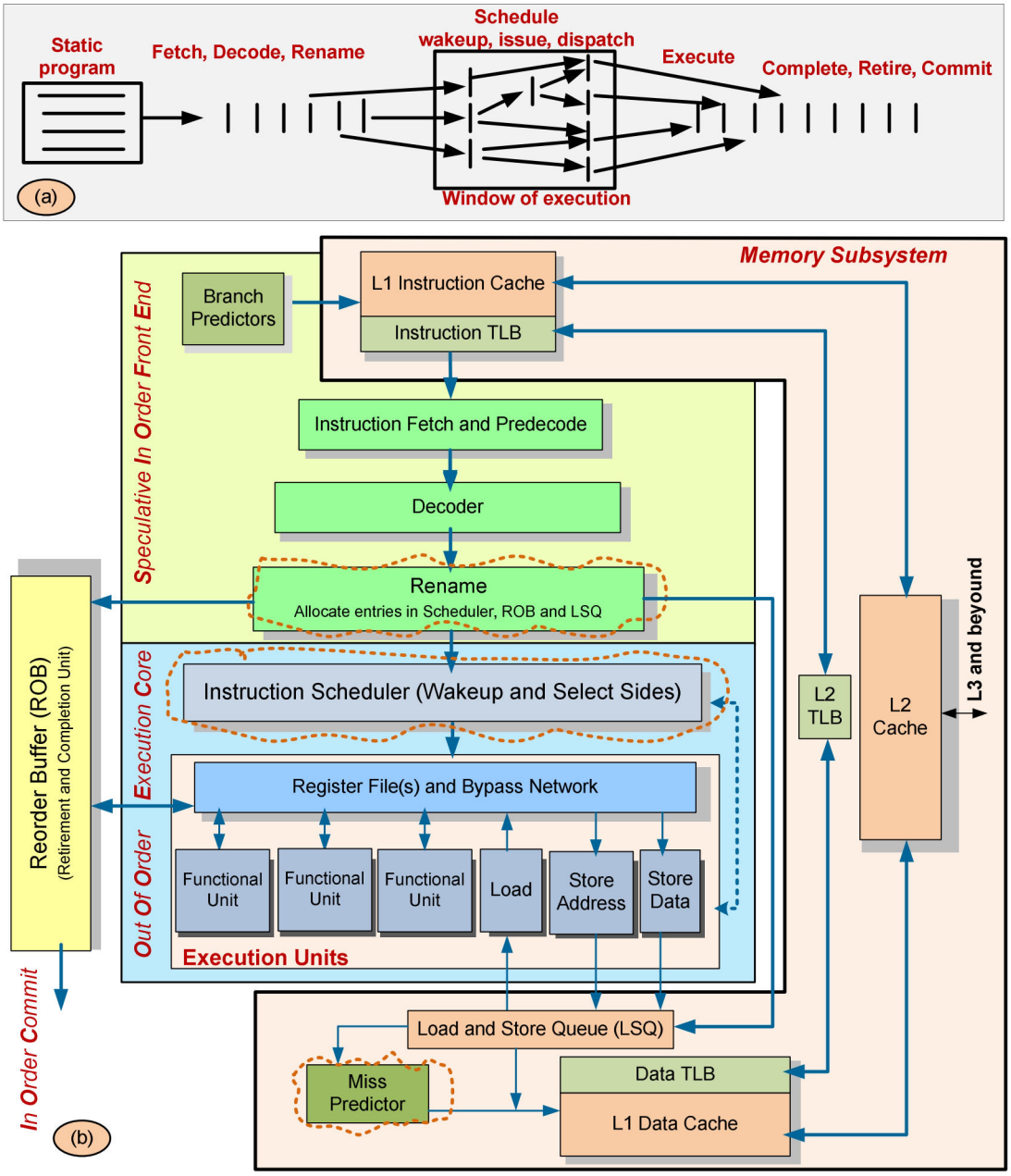


Figure 1-3: Flow-path model, pipeline stages and data-path block diagram of a dynamically-scheduled (out-of-order) superscalar processor.

processor would have to wait for every control-flow instruction to execute before fetching and executing any subsequent instructions (given 20% of the total instructions are control-flow instructions, the amount of ILP that can be extracted will be severely limited). Other forms of speculation are possible, including hit speculation, memory dependency speculation, and value speculation. For example, if instructions are allowed to execute before a preceding load, these instructions are executed speculatively because the load may cause a page fault. Specifically, latency of the load, which depends on the location of the data in memory hierarchy, can be determined by executing the load instruction. Instead of waiting until the latency of the load is known, which introduces pipeline stalls and degrades performance, processors can speculate whether the load will hit or miss the cache and schedule the instructions after the load according to the prediction. For instance, using CBF-based miss predictors, a processor can achieve 99.7% of the *instruction per cycle* (IPC) performance achievable with a perfect predictor [54].

Figure 1-3(a) shows the high-level operation of a typical dynamically-scheduled, superscalar processor. First, the processor fetches instructions following the original, possibly predicted, sequential order. Once these instructions are fetched, the processor checks for data dependencies, removes false dependencies and identifies true ones. Instructions then proceed, in the original sequential order, into the scheduling stage where they wait until all their true dependencies are satisfied. Once this happens, an instruction can request for execution. Instructions may execute and produce results in an order that may or may not be different than the original sequential order. The processor may execute multiple instructions in parallel. Eventually, instructions are *committed* in order. An instruction is committed when its execution would have completed successfully in a processor that executes instructions sequentially. This commit processing allows processors to appear to behave like a processor executing instructions sequentially. In addition, in-order commit processing allows processors to recover from incorrect speculation because before an instruction is committed, it can be safely discarded. An instruction can commit when it has produced its result and all preceding instructions have committed as well.

Various implementations exist for dynamically-scheduled superscalar processors [55] [61]; without the loss of generality, however, we focus on the generic model shown in Figure 1-3 (b)

to review the key components of the modern processors. Depicted in Figure 1-3, superscalar pipelines typically consists of the following logical stages that may or may not correspond to physical stages: fetch, decode, rename, issue, execute, complete, and retirement/write-back.

The processor fetches multiple instructions from the instruction cache. Upon fetching a conditional branch, the processor uses a branch predictor to speculate the most likely execution path that the branch would take, and then continues fetching the instructions along the predicted path. On mispredictions, the processor discards all incorrect instructions and resumes execution by fetching instructions from the correct path. After fetching the instructions, the processor decodes them and sends them for renaming. During renaming, a unique physical register is allocated for the destination of each instruction; these physical register names are dynamically mapped into the architectural register names used by the instructions using a register alias table or RAT (see Section 4.1). This process eliminates all false data dependencies to help exploit ILP. Before a physical register is assigned to the destination register of an instruction, its source register operands are renamed to the physical registers as renamed by the preceding instructions. An instruction being assigned a new physical register records the previous mapping for this register in the reorder buffer (ROB) so that the processor can recover from incorrect speculation.

After renaming, the instructions enter the out-of-order processing stage by allocating entries for the instructions in the scheduler/window and ROB/retirement buffer. The scheduler is loaded with the instructions in order and issues them to the functional units potentially out of program order. Instructions in the scheduler are free from control dependencies due to branch prediction, and free from name dependencies due to register renaming; hence, only true data dependencies and structural conflicts remain to be resolved by the scheduler (detailed in Section 5.1). Instructions wait in the scheduler for their source operands and the appropriate functional unit to become available. Schedulers can be either unified or distributed (e.g., reservation stations that partition the scheduler/instruction window by functional units). For memory instructions, including loads from and stores to the memory, entries are also allocated in the *load-store queue* (LSQ). The LSQ schedules loads and stores for execution. Its task is more complicated than that of the preceding scheduler because the dependencies between loads and stores are unknown until the instructions calculate their addresses.

Various datapaths read source register values at different stages [55]. In one option, used by the latest processors, register values are not read when instructions enter the scheduler. Instead, they are read while the instructions are dispatched to the execution units (in this case, scheduler entries only keep source registers' addresses not their values). When the functional unit finishes the execution of an instruction, the instruction result is written into the corresponding physical register and is forwarded to the instructions waiting in the scheduler. To ensure in-order commit, the ROB buffers the instructions as they may finish execution out of order, and then retires the instructions in order. The ROB is also used to recover from incorrect speculation, by restoring the RAT to the state it had prior to renaming incorrectly-speculated instructions (Section 4.1.4).

1.3 Thesis Contributions

This thesis studies three processor components: *CBFs*, R*ATs* and *CMSs*. CBFs optimize membership tests, commonly used in predictors (e.g., cache miss predictor), RAT is the core of the renaming stage, and CMS is an instruction scheduler's implementation. Specifically, this thesis improves the state-of-the-art by *(i)* proposing a novel physical-level implementation for CBFs, *(ii)* studying tradeoffs among RAT GC count and RAT latency, affecting overall processor performance, and *(iii)* investigating CMS and its related logic at the physical level. For these components, this thesis also presents simple, empirical latency and energy models as well as detailed, analytical models that can be adapted for newer fabrication technologies.

1.3.1 Hardware Counting Bloom Filters

Whether a specific block exists in the cache or not is one of the examples of membership tests in processors. By providing a definite answer for most but not all membership tests, hardware *counting bloom filters* (CBFs) improve upon the power, speed and complexity of various processor components. For example, CBFs have been used to improve performance and power in snoop-coherent multi-processor or multi-core systems [44] [45]. CBFs have also been used to improve the scalability of load/store scheduling queues [60] and to reduce instruction replays by assisting in early miss determination at the L1 data cache [54]. In these applications, CBFs help eliminate broadcasts over the interconnection network in multiprocessor systems [44]; CBFs also help reduce accesses to much larger and thus slower and power-hungry content addressable memories (CAMs) [54] [60], or cache tag arrays [44] [45] [54].

CBFs help improve the energy and speed of membership tests by providing a definite answer for most, but not necessarily all, membership tests. The CBF does not replace entirely the underlying conventional mechanism, but it dynamically bypasses the conventional mechanism, which can be slow and power hungry, as frequently as possible. Accordingly, the benefits of utilizing CBFs depend on two factors. The first factor is how frequently a CBF can be used. Architectural techniques and application behavior determine how many membership tests can be serviced by the CBF. The second factor is the energy and latency characteristics of the CBF itself. The more membership tests are serviced by the CBF *alone* and the more speed- and energy-efficient the CBF implementation is, the higher the benefits. Our work focuses on the second factor as it investigates implementations of a CBF that improve its speed and energy characteristics.

This work is the first to study CBF's physical-level implementations, optimizations and modeling with the following contributions: First, this work proposes L-CBF, a novel, energy- and speed-efficient implementation for CBFs. L-CBF uses an array of up/down linear feedback shift registers and local zero detectors. Second, this work compares the energy, latency and area of L-CBF and S-CBF, a straightforward SRAM-based implementation for CBFs [45], using their full-custom layouts in a 0.13 μm fabrication technology. We demonstrate that depending on the operation type, L-CBF compared to S-CBF is 3.7x or 1.6x faster and requires 2.3x or 1.4x less energy. Third, this work presents analytical models for L-CBF's latency and energy and compares the model estimations against simulation results. These models can estimate energy and latency of various CBF organizations early in the design stage during architectural-level explorations where physical-level implementation either is impossible to develop, or is unaffordable due to time and/or cost constraints. Comparisons show that the model estimations are within 5% and 10% of Spectre™ simulation results for latency and energy respectively.

1.3.2 Checkpointed, Superscalar Register Alias Tables

The register alias table (RAT), the core of register renaming, is a performance-critical component of modern dynamically-scheduled processors. The RAT is considered performance-critical since it is read and updated by almost all instructions in order as they are decoded. Hence, the RAT must operate at the processor's frequency or it must be pipelined. The complexity and size of the RAT, and hence its latency and energy vary as a function of several architectural parameters

such as issue width (IW) and window size (WS). RAT complexity is increased further by the use of speculation, control flow or otherwise. On mispeculations, the RAT content must be restored not to contain any of the mappings introduced by incorrectly-speculated instructions. Accordingly, the RAT incorporates a set of global checkpoints (GCs) to recover from mispeculations. A GC is a complete snapshot of all relevant processor state including the RAT, and it is used to recover from control-flow mispeculations. Recovery at an instruction using a GC is "instantaneous", i.e., it requires a fixed, low latency.

Recent work assume that incorporating many GCs in the RAT increases latency unacceptably [3] [4] [20]. However, previous studies on RAT checkpointing did not quantify how RAT latency is affected by GCs. Instead, they ignored RAT latency and focused solely on instruction per cycle (IPC) performance evaluations assuming that reducing the number of GCs is imperative. This is the first work to study quantitatively how RAT latency and energy vary as a function of the number of GCs (NoGCs). This information is imperative to understand the actual performance, energy and GC count tradeoffs and to determine whether existing state-of-the-art solutions work sufficiently well, or whether further innovations are needed.

Accordingly, this work complements previous studies by investigating performance and energy using full-custom checkpointed RAT implementations in a 0.13 μm technology. This work studies the actual performance and energy impact of introducing more GCs considering state-of-the-art confidence-based methods for allocating GCs selectively [3] [4] [5] [31]. Our work demonstrates that ignoring the actual RAT latency incorrectly predicts performance. In fact, two components contribute to determining performance. First, as more GCs are introduced, fewer cycles are spent recovering from mispeculations, hence improving IPC. Second, introducing more GCs increases RAT latency, and hence increases clock period and decreases performance. In most cases, using very few GCs (e.g., four) leads to the optimal performance. Our work justifies those RAT checkpointing techniques that try to sustain performance with as few GCs as possible and highlights the importance of GC management methods that help achieve this goal.
Commonly-used RAT implementations are based on SRAM or CAM structures. We refer to them as sRAT and cRAT respectively. This work also studies the latency and energy characteristics of both sRAT and cRAT implementations, and compares their energy and latency

variation trends. We have found that these two implementation styles exhibit different scalability trends. Typically, sRAT is faster and consumes less energy than cRAT. The sRAT is less sensitive to an increase in WS (or equivalently of the number of physical registers) or IW. The difference between sRAT and cRAT grows larger with increasing the number of GCs (NoGCs). For example, 4-way sRAT lookups require 33% less energy and are 12.1% faster than cRAT lookups when the number of physical registers is 128. The differences grow to 416% for energy and 34.6% for latency when the number of physical registers increases to 512. On the other hand, cRAT is less sensitive to increasing NoGCs. When NoGCs passes a limit, cRAT becomes faster than its equivalent sRAT. Motivated by this result and by the existence of commercial designs that use cRAT with large number of GCs [33], we propose an energy optimization where only the entries containing the latest mappings for an architectural register remain active during lookups. The energy savings mostly are a function of the number physical registers. For instance, for a 128-entry cRAT, the energy per lookup operation is reduced by 40%.

Based on the full-custom layouts, we also present analytical models for sRAT latency and energy. These models can predict the variation of RAT latency and energy as a function of several parameters, thereby providing insight into the latency and energy of various RAT alternatives during architectural-level exploration. Our model estimations are within 6.4% and 11.6% of Spectre™ simulation results for latency and energy respectively.

The contributions of this work are as follows: *(i)* It presents full-custom implementations for the checkpointed sRATs and cRATs of 4-way and 8-way dynamically-scheduled, superscalar processors in a 130 nm fabrication technology. *(ii)* For all RAT operations, it quantitatively determines sRAT's and cRAT's latency and energy as a function of IW, WS and NoGCs. *(iii)* Using architectural-level simulations, this work estimates how performance is affected by sRAT latency for two different checkpointed sRAT implementations when a state-of-the-art selective GC allocation policy is taken into consideration [3] [5] [4]. *(iv)* It presents analytical models for sRAT latency and energy and compares the model estimations against circuit simulation results. *(v)* It compares the energy and latency variation trends of sRAT and cRAT. *(vi)* It proposes an energy optimization for cRAT.

1.3.3 Compacted Matrix Schedulers

Dynamically-scheduled processors exploit ILP by buffering instructions and scheduling them potentially out of the program order. The scheduler, responsible for the buffering and scheduling, typically comprises wakeup and select stages. Instructions wait in the wakeup stage to become ready, i.e., all their input operands become available. The select stage involves picking a set of ready instruction for execution taking into consideration resource constraints. The scheduling loop formed between the wakeup and select sides is critical for performance. The scheduling loop's delay is a function of the instruction scheduler size (window size or WS) and to a lesser extent of the IW. The scheduling loop's delay is a function of WS (or scheduler size). Larger schedulers issuing more instructions per cycle could improve the IPC completion rate. However, commercial processors do not use large schedulers since they are slow and they tend to limit processor's clock period, resulting in lower overall performance. For example, recent Intel and AMD desktop/server processors have integer scheduler sizes of 24 to 32 entries [58].

The wakeup side holds instructions waiting for inputs yet to be produced, matches waiting instructions with incoming results, and identifies ready-for-execution instructions (all inputs are available). The wakeup's matching functionality can be implemented using CAMs or dependency matrices [58]. The matrix-based implementation is more speed- and energy-efficient than the CAM-based implementation. Compacted matrix schedulers use an indirection table (wakeup allocation table or WAT) to reduce the matrix width, and hence wakeup latency.

While arguments have been made in support of the speed and scalability advantages of compacted matrix-schedulers, neither actual measurements from physical-level implementations nor models have not been reported. This is the first work to fill this gap by investigating the delay and energy variation of compacted matrix schedulers and the checkpointed WAT as a function of IW, WS and NoGC. This work uses full-custom implementations in a commercial 90 nm fabrication technology. This physical-level study is useful in exposing design inefficiencies or optimizations opportunities. As an example of these opportunities, this work proposes an energy optimization that throttles pre-charging. For 32×20 and 64×20 matrices, this optimization reduces energy by 10% and 18% respectively.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 provides background on modeling and design methodology. Section 2.1 briefly discusses high-level, architectural models and discusses their shortcomings. Section 2.2 reviews previous work that focused on developing models for processor components to propose architectural- and/or physical-level optimizations. Section 2.3.3 reviews existing analytical and empirical approaches to develop models.

Chapter 3 focuses on CBF implementations, modeling and optimizations. Section 3.1 reviews CBFs and their previously-assumed implementation. Section 3.2 presents L-CBF, our novel implementation for CBFs. Section 3.3 discusses the analytical models for L-CBF. Section 3.4 presents the evaluation results, and finally, Section 3.5 summarizes our findings.

Chapter 4 focuses on sRAT and cRAT implementations, modeling and optimizations. Section 4.1 provides background on RAT implementations and checkpointing. Section 4.2 discusses physical-level implementations for checkpointed sRAT and cRAT respectively. Section 4.3 presents analytical models for sRAT. Section 4.4 presents the physical-level and architecture-level evaluation results. This section compares the simulation results against the model estimations for sRAT. This section also compares the latency and energy variation trends of sRAT and cRAT. Section 4.5 summarizes the key points of this work.

Chapter 5 focuses on the modeling and optimization of CMS and its related logic. Section 5.1 provides background on CAM-based and matrix-based schedulers. Section 5.2 reviews the matrix and WAT. Section 5.3 discusses physical-level implementation details. Section 5.4 presents the evaluation results, and finally Section 5.5 presents the key results of this work.

Chapter 6 concludes this thesis and highlights its key findings.

2. Background: Design Methodology and Related Work

This chapter provides background on modeling and design methodology. This general review chapter is divided into three parts. The first part reviews Wattch, representative of high-level power estimator, and discusses its shortcomings [12]. This discussion motivates this work by illustrating the improvement opportunities for component modeling based on physical-level implementations. The second part reviews other work that focused on developing analytical models for processor components. These models were used to evaluate the hardware complexity characteristics of various components and to propose architectural- and/or physical-level optimizations. The third part reviews existing analytical and empirical approaches to develop models for processor components. These models characterize component's latency and energy and are used in high-level architectural exploration.

2.1 Architectural-Level Models

Quick estimation of power and performance is the goal of many high-level power-performance simulators. The principles underline these simulators are common. Accordingly, we limit our review to Wattch, a representative of high-level power estimators [12] [71] [67].

Depicted in Figure 2-1, Wattch relies on the Simplescalar [16] , which is a cycle-accurate performance simulator, to estimate per cycle component-level activity factors (α). These activity factors are then used in dynamic power estimation. Dynamic power ($P_{dynamic} = \alpha \times f \times C \times V_{dd}^{2}$) depends on the frequency (f), the capacitance(C), the supply voltage (V_{DD}) and the activity factor (α). Wattch's analytical power models are equations controlled by technology-specific per-unit-length capacitances for metals, gate and diffusion in addition to the structural geometry parameters of the modeled component. Wattch's power models fall into two categories: *(i)*

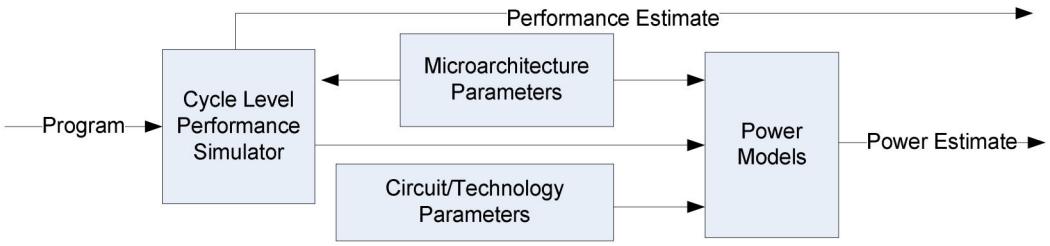


Figure 2-1: High-level block diagram of an architectural-level power estimator

analytical models in form of equations describing microarchitectural building blocks for specific circuit classes—e.g., SRAM; *(ii)* technologically-scaled power measurements derived from the published data for difficult-to-analyze, irregular logic structures—e.g., functional units. The analytical models relied on older CACTI's power models that were developed based on custom implementations of specific circuit classes in old fabrication technologies (0.8µm) [69]. A scaling model was used to convert the model's parameters for the newer technologies.

The limitations of high-level power and/or performance estimators are as follows. Wattch does not model interconnect capacitances accurately. In capacitance modeling, Wattch neglects overlap capacitances, which could be comparable to other capacitances in the newer fabrication technologies. Wattch models also ignore the dependency of the transistor's capacitances on the supply voltage. Wattch focuses only on the dynamic power and neglects secondary effects such as static leakage power. However, for fabrication technologies below 0.13 µm, leakage power is significant and largely depends on the circuit state [40]. Hence, obtaining a reliable estimation of the leakage power without physical-level simulations is difficult. Another limitation is that Wattch does not model the dependency of the power consumption on temperature, while leakage power increases rapidly as temperature rises.

Wattch focuses on specific processor architecture and assumes specific implementations for the processor components. Hence, to evaluate a different architecture or implementation, Wattch models need to be appropriately modified. Because Wattch does not consider all possible circuit-level implementations for a component, speed- and/or power-efficient alternatives cannot be investigated without introducing a new model for each alternative design. While in some cases modifying the existing models is possible, in other cases developing new models is required. For instance, an alternative implementation for a sense amplifier is shown to be 29% more power efficient than the one used in the Wattch based on older versions of CACTI [12]. From the architectural optimization perspective, Wattch needs modifications to accommodate new customized architectures. As another example, Parikh et al. proposed a new design for branch predictor that required banking of the array structure [52] [53]. This design required dividing the decoder to row and column decoders, whereas the Wattch model considered a monolithic decoder. Accordingly, this work developed its own model for its proposed branch predictor so that the underlying array structure has separate row and column decoders.

Tools like Wattch must be extended to evaluate various architectural- and/or physical-level alternatives/optimizations. To validate such optimizations for a component, a new or a modified physical-level implementation is required. In such cases, developing a customized physical-level implementation is inevitable to recognize design inefficiencies. This thesis complements such tools as it studies several processors' components that no physical-level investigations or models have been reported for them. In addition, adapting existing models (e.g., Wattch models) for modeling these components is not feasible.

2.2 Related Work: Quantifying Hardware Complexity of Modern High-Performance Processors

Paracharla [49]-[51] and Zyuban [72]-[74] developed simple analytical models to study the hardware complexity characteristics of out-of-order, superscalar processors' components. Based on these models, they proposed new, speed- and/or power-efficient processor architectures.

Paracharla et al. developed latency analytical models for several latency-critical components of a generic superscalar microarchitecture [49]-[51]. For this study, a component is considered to be latency-critical if its latency is a function of IW, WS, interconnect delay, or a combination thereof. The goal of this study was to identify the components whose latencies would determine the cycle time for future processors. Simple latency analytical models were developed expressing the delay of each component as a function of several architectural parameters. In addition, this work investigated how component latencies scale as feature size shrinks and interconnect delay becomes more prominent. This work concluded that the scheduler and the data bypass logic tend to be increasingly critical for wider superscalars and larger windows and smaller feature sizes. The key reason for the poor scalability was found to be that both scheduler and data bypass logic rely on broadcasting values on long wires. For smaller feature size, interconnect delay was found to increasingly dominate the total delay.

Zyuban and Kogge developed analytical models for several energy-critical components of a generic superscalar microarchitecture [72]-[74]. In this study, a component is considered to be energy-critical if its energy is expected to grow as processor scale to exploit more ILP. These models were used to estimate the lower bound on the energy dissipation that can be achieved or

approached by various circuit-level techniques. Their work demonstrated that the most energy-critical components of a typical processor are the multi-ported register file, the instruction scheduler, the memory disambiguation unit and the bypass logic.

Our work complements previous work in that it studies components that no physical implementation or models have been reported for them.

2.3 Design and Implementation Methodology

This section discusses our model development methodology. Our methodology consists of several steps: First, we develop a physical (circuit and layout) level implementation for the component in a state-of-the-art technology. We apply various delay and/or power optimizations to the circuit. Then, we exploit a combination of analytical and empirical modeling approaches to develop models capable of estimating the delay and power for the component. These models are in form of equations driven by several organizational and technology parameters. Through the process of developing physical-level implementation and analytical models, we gain insight into the physical-level characteristics of the component. Having both architectural- and physical-level evaluation results for the component, we can *(i)* explore various architectural alternatives, *(ii)* identify inefficiencies in the existing designs, and *(iii)* propose an alternative speed- and/or energy-efficient implementation.

2.3.1 Physical-Level Implementation

We first develop a transistor-level implementation and layout for the component in an available state-of-the-art fabrication technology. Before developing a circuit-level implementation, we study existing, published implementations. Once a candidate implementation is selected, the transistors are sized based on the optimization objective, which can be power, latency, or a combination thereof. After the circuit implementation phase, we develop a custom physical layout. The physical layout is required to gain insight into the overall circuit hardware complexity (speed, energy and area). Furthermore, the physical layout determines the transistor's placement and characteristics, the parasitic capacitances and resistances, and the area. The information gathered during the design is then used to develop the analytical models.

All physical design is done using the Cadence™ tool set. For physical-level simulations Spectre™ is used. To compare alternative designs that are mostly similar, we take advantage of faster but less accurate simulation tools such as Nanosim Synopsys™.

2.3.2 Physical-Level Evaluation and Optimizations

Using our models (Section 2.3.3), we investigate the latency and energy variations as a function of architectural-level parameters (e.g., issue width). Through the physical-level implementation and modeling phases, the design inefficiencies of existing designs are recognized. Being aware of these inefficiencies, we suggest alternative speed- and/or energy-efficient physical- and/or architectural-level implementations.

2.3.3 Developing Analytical Latency and Energy Model

After the physical implementation, we develop latency and energy models. These models are derived using a bottom-up modeling methodology that utilizes physical-level information. We use RC circuit analysis to develop analytical models capturing the dependency of the latency and energy on several parameters (e.g., for an array-like structure, the parameters would be the number of entries and the entry width). These models are equations driven by fabrication technology and physical-level parameters. The physical characteristics of a component (e.g., entry count for an array structure) directly relate to the architectural parameters (e.g., issue width). Hence, these models can be used to determine the variation trend of the latency and energy as a function of these architectural parameters. To determine the accuracy of model predictions, we compare physical-level simulation results against analytical model estimations.

2.3.3.1 Analytical-Empirical Modeling Approaches

This section reviews existing modeling approaches. High-level latency and energy models enable computer architects to explore early-stage microarchitecture options from power- and/or performance-efficiency perspective [13] [14] [15]. Latency and energy models can be developed using the analytical and/or empirical approaches, which are discussed in this section.

Analytical Models: Analytical models are in form of equations driven by high-level organizational parameters, physical-level design parameters and technology-specific parameters.

The high-level organizational parameters can be a function of architectural-level design parameters such as IW, WS, number of physical registers, number of pipeline stages, misprediction penalties, cache geometry parameters or queue/buffer sizes.

As an example of analytical models, consider dynamic power analytical models, which are developed based on the internal nodes' switched capacitances. For instance, (1) calculates SRAM's bitline energy where $C_{bitline}$ is the bitline capacitance and V_{swing} is the bitline voltage swing. For a simple SRAM, the bitline capacitance, given by (2), includes N stacked pass transistor's diffusion capacitances, pre-charge circuit capacitance, bitline interconnect capacitance and column-select circuit capacitance. Figure 2-2 depicts the transistor-level implementation of an SRAM column.

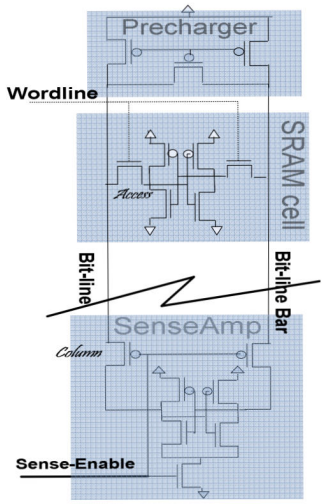


Figure 2-2: An SRAM Column

$$E_{dynamic} = C_{bitline} \times V_{dd} \times V_{swing} \quad (1)$$

$$C_{bitline} = C_{diffution\ access} \times N + 2 \times C_{diffution\ precharge} + C_{diffiution\ column} + C_{wire} \quad (2)$$

$$C_{diffution} = AD \times C_J + PD \times C_{JW} \quad (3)$$

$$C_{gate} = C_{ox} \times AD \quad (4)$$

Although flexible and fast, analytical-model estimations may have large errors when compared to the physical-level simulation results for deep sub-micron feature sizes due to several factors. Inaccurate capacitance estimation is one source of the error. The analytical models utilize (3) and (4) to estimate transistor diffusion and gate capacitances, where AD (W×L) is the drain area and PD (W+2×L) is the drain perimeter calculated based on the transistor geometry (W, L). These formulas neglect overlap capacitance, which could be comparable to the other capacitances in some cases. Also, these formulas disregard the voltage dependency of the capacitance. Node capacitances, including overlap and node-to-node coupling capacitances, also depend significantly on circuit design, hence their estimation is difficult without circuit simulations.

Analytical power models usually neglect direct-path current whose estimation is difficult because of the dependency on the circuit design. Most analytical power models exclusively consider dynamic power consumption and neglect secondary effects such as static leakage power, which are significant for fabrication technologies smaller than 130 nm. Analytical power models for leakage power are typically based on formulas for sub-threshold and gate leakage currents [36] as estimated by (5). Several of this formula's parameters depend on the circuit topology and state. Specifically, β and n are empirically-derived parameters. Hence, an accurate perspective of leakage current requires circuit simulations.

$$I_{leak} = \beta \times e^{b\times (V_{dd}-V_{dd0})} \times V_t^2 \times (1- e^{-Vdd/Vt}) \times e^{(-|V_{th}|-V_{off}\times n\times Vt)} \quad (5)$$

Power is also closely related to temperature. In particular, leakage power increases as temperature rises. Not all analytical models take temperature into account, thus leading to inaccurate power estimations. To improve accuracy, the analytical dynamic power models can be calibrated for leakage estimations with circuit simulations [22].

Empirical Models: One category of empirical models use detailed physical-level simulation data (measurements) from prior processors to develop models for the next-generation processors using technology scaling. Another category of models rely on the physical-level simulation of the component implementations available from previously-implemented processors. Models of this type require significant simulation time, and thus are expensive or impractical to use. In addition, empirical models based on the previous implementations in older fabrication technologies do not scale well to newer technologies. These models can be a good choice when many components tend to be reused in newer designs with relatively small changes. Hence, flexibility is a major shortcoming of the empirical models, especially if a new fabrication technology or a new microarchitecture is being proposed. Empirical models are often based on a specific circuit design, while analytical models can be extended for new circuit design styles.

Hybrid Analytical-Empirical Models: The advantages of the analytical models, which are also the disadvantages of the empirical models, are efficiency, flexibility and scalability. However, the analytical models unlike the empirical models are not very accurate.

A hybrid analytical-empirical approach can have the advantages of both analytical and empirical approaches. The hybrid modeling approach divides the circuit into small building blocks and extracts latency and power information from the detailed physical-level simulation for these blocks in every new technology [37]. Detailed physical-level simulations on small blocks can capture relevant technology information about the new fabrication technology. Given this empirical information, analytical formulas employ these building blocks' information to model the delay and power of larger structures. These hybrid analytical-empirical models can have accuracy comparable to physical-level simulation because these models use circuit simulations for their small building blocks, thereby taking into consideration the characteristics of the implementation fabrication technology.

2.3.3.2 Memory Array Models

Several work proposed models for SRAM/CAM-based memory arrays that are the base of many processor components such as register files, caches, translation look-aside buffers (TLBs), queues and buffers. Zyuban et al. proposed analytical power models for register files [72]. Mamidipaka et al. provided a high-level power estimation tool (IDAP) for SRAMs considering various circuit styles using physical-level parameter inputs (cell area, sense amplifier design, etc) [38] [39]. Agrawal et al. presented power and delay models for CAMs [2]. Many work focused on developing energy analytical models for caches [29] [32] [35] [23].

A popular tool for estimating cache area, performance and energy is CACTI. Over the years, CACTI has been enhanced for more accurate delay, power and area estimation for specific implementations of caches at the microarchitectural-level. CACTI is also used in various system-level tools such as Wattch [12], SimplePower [67] and HotLeakage [71] for design space exploration. Before CACTI, analytical models were proposed to estimate cache area by Mulder et al. [47] and to estimate cache access time by Wada et al. [68]. CACTI 2.0 included power models to evaluate power and access time tradeoffs in different cache configurations [56]. CACTI 2.0 added modeling support for fully-associative caches, a power model, technology scaling, multi-ported caches and improved tag comparison circuits. CACTI 3.0 included modeling support for the cache area estimations, caches with independently-addressed banks, reduced sense-amplifier power dissipation [62]. eCACTI addressed some limitations of power models used in CACTI 3.0 and added leakage power models [40]. CACTI 4.0 added leakage

power models and updated the basic circuit structures and device parameters to reflect recent advances in semiconductor scaling. CACTI 4.0 provided support for user-defined tag and data widths, a serial tag and data access option for low power and modeling of memory structures without tags [65]. In deep-submicron fabrication technologies, device and process parameter scaling are non-linear, hence the base technology modeling has been changed in CACTI 5.0 from simple linear scaling used in the original CACTI 0.8 μm technology to the scaling based on the ITRS roadmap [66]. Additionally, CACTI 5.0 models dynamic random access memories (DRAMs) and provides the possibility of SRAM and DRAM comparisons. Various circuits have also been updated in CACTI 5.0 to adapt to modern design practice.

Most of the discussed models focused on specific reference physical-level implementations and considered bottom-up modeling approach. Top-down approach is another possible approach for developing models. For instance, Agrawal et al. in [1] develops a latency model for SRAMs, given by (6), exploiting a non-linear regression technique [59] [19] on almost 60 published industrial and academic SRAM implementations. Selected implementations were targeted feature sizes ranging from 0.8 μm to 0.065 μm. The SRAM sizes ranged from 1K bit to 70M bit and included one or two ports.

$$\text{delay (ns)} = 0.27\times(\text{tech})^{1.38} \times (\text{bits})^{0.25}\times (\text{port})^{1.30} + 1.05 \qquad (6)$$

For developing our detailed analytical models, we followed the methodology of high-level tools such as CACTI. For developing our simplified, empirical models, however, we used the extrapolation- and regression-based techniques.

2.4 Summary

This chapter provides background on the component modeling approaches as well as design methodology. Additionally, it reviews previous work with similar objectives to highlight shortcomings in current modeling and optimization areas. Subsequent chapters focus on specific components, and for each component enumerate the contributions made regarding the modeling and optimization.

3. Counting Bloom Filters

By providing a definite answer for most, but not necessarily all, membership tests required in various functions within the processors, hardware *counting bloom filters* (CBFs) improve upon the power, speed and complexity of various processor components. An increasing number of architectural techniques rely on CBFs to improve upon the power, latency, and complexity of various processor structures. For example, CBFs have been used to improve performance and power in snoop-coherent multi-processor or multi-core systems [44] [45]. CBFs have also been utilized to improve the scalability of load/store scheduling queues [60] and to reduce instruction replays by assisting in early miss determination at the L1 data cache [54]. In these applications, CBFs help eliminate broadcasts over the interconnection network in multi-processor systems [44]; CBFs also help reduce accesses to much larger, and thus much slower and power-hungry content addressable memories [60] [54] or cache tag arrays [44] [45] [54].

In all aforementioned hardware applications, CBFs improve the energy and speed of membership tests (e.g., checking whether a memory block is currently cached [54]). The power and performance advantages of CBFs come at the expense of only partial coverage, i.e., a CBF provides a definite answer for most, but not necessarily all, membership tests. Hence, the CBF does not replace entirely the underlying conventional mechanism (e.g., cache tags), but it dynamically bypasses the conventional mechanism, which can be slow and power hungry, as frequently as possible. Accordingly, the benefits obtained through the use of CBFs depend on two factors. The first factor is how frequently the CBF will be successful at eliminating accesses to the underlying conventional mechanism. Architectural techniques and application behavior determine how many membership tests can be serviced by the CBF. The second factor is the speed and energy characteristics of the CBF itself. The more membership tests are serviced by the CBF 'alone' and the more speed and energy efficient the CBF is, the higher the benefits.

This work focuses exclusively on the second factor and investigates CBF implementations that improve its energy and latency characteristics. Previous work assumed a straightforward SRAM-based implementation that we will refer to it as S-CBF [45]. The main contribution of this work is the introduction of the L-CBF; L-CBF is a novel, energy- and speed-efficient CBF

implementation that improves upon S-CBF implementation. L-CBF utilizes an array of up/down linear feedback shift registers (LFSRs) and local zero detectors. We investigate the energy, latency and area characteristics of both L-CBF and S-CBF implementations in a commercial 0.13 μm fabrication technology. We demonstrate that depending on the type of operation, L-CBF compared to S-CBF is 3.7x or 1.6x faster and consumes 2.3x or 1.4x less energy. We also present analytical energy and latency models for the L-CBF. These models can estimate energy and latency of various CBF organizations early in the design stage during architectural-level explorations. These explorations are performed well before the physical-level implementation phase. Comparisons show that the model estimations are within 5% and 10% of Spectre™ circuit simulation results for latency and energy respectively.

The contributions of our work are as follows: *(i)* It proposes a novel, energy- and speed-efficient implementation for CBFs, L-CBF; *(ii)* It compares the energy, latency and area of two CBF implementations, L-CBF and S-CBF, using their full-custom layouts in 0.13 μm fabrication technology; *(iii)* It presents analytical latency and energy models for L-CBF and compares the model estimations against the circuit simulation results to determine the model's accuracy.

The rest of this chapter is organized as follows: Section 3.1 reviews CBFs and their previously assumed implementation, S-CBF. Section 3.2 presents L-CBF, our novel implementation. Section 3.3 discusses the analytical latency and energy models for the L-CBF implementation. Section 3.4 presents the experimental results. Finally, Section 3.5 summarizes our findings.

3.1 Background

This section reviews the characteristics and functionality of CBFs. Additionally, this section discusses the previously-assumed implementation for the CBFs. This discussion serves as motivation for the L-CBF design explained in the next section.

3.1.1 CBF As A Black Box

Depicted in Figure 3-1, a CBF is conceptually an array of counts indexed via a hash function of the element under membership test. CBFs support three operations: *(i)* increment count (INC); *(ii)* decrement count (DEC); and, *(iii)* test if the count is zero (PROBE). The first two operations

increment or decrement the corresponding count by one, and the third one checks if the count is zero and returns true or false (single-bit output). We will refer to the first two operations as updates and to the third one as a probe. A CBF is characterized by the number of entries and the width of count per entry.

3.1.2 CBF Characteristics

Membership tests using CBFs are performed by probe operations. In response to a membership test, a CBF provides one of these two answers: *(i)*"definite no", indicating that the element in question is definitely not a member of the large set, and *(ii)* "I don't know", implying that the CBF cannot assist in the membership test, and hence the large set must be searched.

The CBF can produce the desired answer to a membership test much faster and reduce power consumption on two conditions: First, if accessing the CBF is significantly faster and requires much less energy than accessing the large set. Second, if most membership tests are serviced by the CBF. The second condition is influenced primarily by application behavior. For instance, when CBF is used as a miss predictor, previous work demonstrated that CBFs can service more than 95% of the L1 cache misses [54].

The CBF uses an imprecise representation of the large set to be searched. Ideally, in the CBF, a separate entry would exist for every element of the set. In this case, the CBF would be capable of precisely representing any set. However, this capability would require keeping a prohibitively large array, thereby negating any benefits. In practice, the CBF is a small array and the element addresses are hashed onto this small array. Due to hashing, multiple addresses may map onto the same array entry. Hence, the CBF constitutes an imprecise representation of the large set's content and represents a superset of the large set. This impreciseness is the reason of the "I don't know" response. Multiple CBFs with different hash functions can be used to improve accuracy.

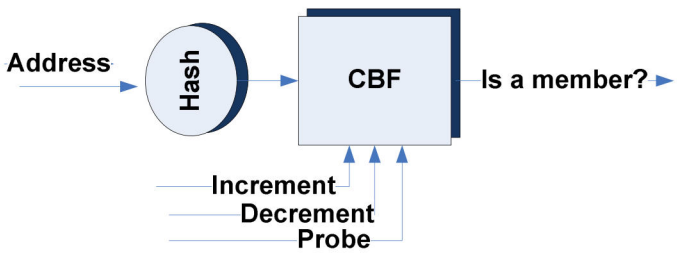


Figure 3-1: CBF as a black box

An ”I don’t know” answer to a membership test incurs power and latency penalty since the actual set (or, simply the set in the rest of the discussion) must be checked as well. The latency penalty occurs if the CBF and the actual set accesses are serialized. This latency penalty can be avoided if we probe the CBF and the set in parallel. In this case, power benefits will be achieved only if we can terminate the in-progress access to the large set once the CBF provides a definite answer. An investigation of these options is beyond the scope of this work. However, in the existing CBF applications, the CBF provides a definite answer in the vast majority of the cases [44] [45] [54] [60]. Hence, using the CBF in series with the conventional membership test mechanism does not, on average, noticeably increase latency.

3.1.3 CBF Functionality

The CBF operates as follows. Initially, all counts are set to zero indicating that the set is empty. When an element is inserted into, or deleted from the set, the corresponding CBF count is incremented or decremented by one. To test whether an element currently exists in the set, we inspect the corresponding CBF count. If the count is zero, the element is definitely not in the set; otherwise, CBF cannot help in membership test, and hence the large set must be searched.

3.1.4 CBF As A Cache Miss Predictor

As an example, we discuss the CBF utility in the early detection of L1 data cache misses [54]. The concepts and implementations are also applicable to other applications.

In this application, the CBF determines if a particular memory block is currently cached in the L1 data cache. Given a block address *A*, the CBF reports whether *A* appears in any of the data cache tags. The CBF provides two possible answers: *(i)* “definite no”: the block is definitely not cached, and *(ii)* “I don’t know”: the block may or may not be cached. In the first case, the CBF determines that this access will result in a miss. Provided that the CBF is much faster and dissipates much less power than the L1 tag arrays, the CBF produces the desired answer much faster and saves power. In the second case, the CBF cannot provide a definite answer and the L1 tags must be checked. This case incurs an extra power penalty because we access both the CBF and L1 tags, and it may incur a latency penalty if the CBF and the L1 tag accesses are serialized. We may avoid this latency penalty by probing the CBF and the L1 tags in parallel, in which case

power benefits will be possible only if we can terminate the in-progress L1 tag access as soon as the CBF provides a definite answer.

Previous work shows that for 95% of the L1 cache misses, the CBF provides a definite answer (an early miss determination), and thus helps consume less power [54]. Early miss determination improves processor performance and reduces power. In particular, in lieu of an early miss determination, processors optimistically assume that all cache accesses will be hits and schedule dependent operations for execution in anticipation of a hit. If the access results in a miss, these dependent instructions need to be stopped from progressing further, an action that takes at least one more cycle to complete. Besides, these cancelled instructions must be stored for re-execution when eventually the actual data arrives. For this reason, high-performance processors include miss predictors. CBFs have been shown to outperform such predictors [54].

The CBF operates as a miss predictor as follows. Initially, all CBF's counters are initialized to zero and the L1 cache is empty. When a block is allocated into or evicted from the L1, the corresponding CBF entry's count is incremented or decremented by one respectively. To test whether *A* currently exists in the L1, we inspect the corresponding CBF count. If the count value is zero, then *A* is definitely not in the L1; otherwise, it is unknown whether *A* is cached or not. The definite "not in the cache" responses allow us to determine misses early.

If CBF were to have a separate entry for each memory address, it would be capable of precisely representing any set of cached addresses, and hence capable of perfect miss detection. However, this preciseness comes at the price of keeping a prohibitively large table negating any benefits. For instance, a table with 32 million entries would be needed for a processor with a 4Gbyte address space and 32-byte cache block size. In practice, the CBF is a small table and the addresses are hashed onto this small table. When a CBF is used as a miss predictor, using a portion of the address and not a more elaborate hash function has been shown to work well for virtual addresses [44] [53] [54]. Multiple CBFs with different hash functions can be used to improve prediction accuracy [53] [54]. Since the same count entry is incremented and decremented on block allocation and eviction respectively, a count can never become negative or

exceed the number of the total cache blocks. Depending on the cache organization and the hash functions used, the maximum count per entry can be much lower than the total number of blocks.

3.1.5 S-CBF: SRAM-Based CBF Implementation

Previous work assumes a CBF implementation consisting of an SRAM array of counts, a shared up/down counter, a zero-comparator/detector and a small controller [45]. We will refer to this implementation as S-CBF. The architecture of S-CBF is depicted in Figure 3-2. Updates are implemented as read-modify-write sequences as follows: *(i)* The count is read from the SRAM; *(ii)* It is adjusted using the counter; and, *(iii)* It is written back to the SRAM. The probe operation comprises an SRAM read followed by a compare with zero using a zero-comparator/detector. A small controller coordinates these actions.

An optimization was proposed to enhance the probe's speed and power by adding an extra bit, Z, to each count [45]. If the count is non-zero, the Z bit is set to false; otherwise, true. Probes can simply inspect Z bits. The Z bits can be implemented as a separate SRAM array, which is faster and consumes much less power. This optimization is applicable to both S-CBF and L-CBF.

3.2 Novel LFSR-Based CBF Implementation: L-CBF

Section 3.4.1 demonstrates quantitatively that much of the energy in S-CBF is consumed on the SRAM's bitlines and wordlines. Additionally, in S-CBF, both latency and energy suffer as updates require two SRAM accesses per operation. The shared counter may increase the latency

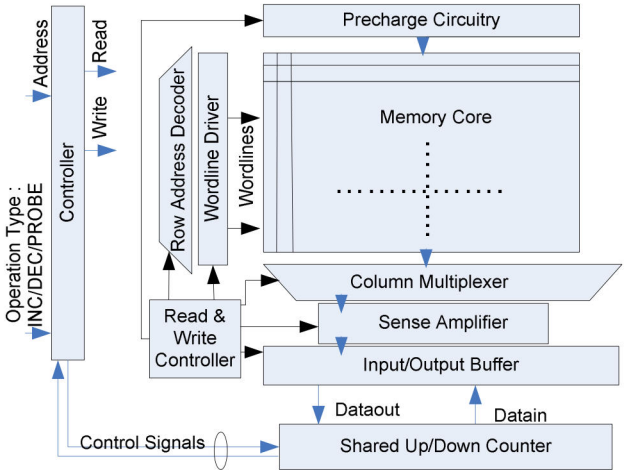


Figure 3-2: S-CBF architecture: an SRAM holds the CBF counts (INC/DEC: read-modify-write sequences, PROBE: read-compare sequences)

and energy further. We could avoid accesses over long bitlines by building an array of up/down counters with local zero detectors. In this way, CBF operations would be localized and propagating read/write values over long bitlines would not be required. L-CBF is such a design. The key insight behind L-CBF is that the actual count values are unimportant, and we only care whether a count is "zero" or "non-zero". Hence, any counter that provides a deterministic up/down sequence can be a choice of counter for the CBFs. The L-CBF's architecture comprises an array of up/down LFSRs with embedded zero detectors. L-CBF employs up/down LFSRs; LFSRs offer a better speed, power and complexity tradeoff than other synchronous up/down counters with the same count sequence length (discussed further in Section 3.2.3).

As Section 3.4 demonstrates, L-CBF is significantly more speed- and power-efficient than S-CBF at the expense of more area. The area penalty is a minor concern in modern processor designs given the abundance of on-chip resources and the small size of CBF compared to most other processor structures (e.g., caches and branch predictors). The rest of this section reviews LFSRs, the construction of up/down (reversible) LFSRs and presents the architecture of L-CBF.

3.2.1 Linear Feedback Shift Registers (LFSRs)

A maximum-length n-bit LFSR sequences through 2^n-1 states consisting of all possible code permutations except one. The LFSR comprises a shift register and a few embedded XNOR gates fed by a feedback loop. The defining parameters of an LFSR are as follows:

- The width/size of the LFSR (it is equal to the number of bits in the shift register).
- The number and positions of taps (taps are special locations in the LFSR that have a connection with the feedback loop).
- The initial state, any value except a specific one (e.g., all ones for XNOR feedback).

Without the loss of generality, we restrict our attention to the Galois implementation of LFSRs [6]. State transitions proceed as follows. The non-tapped bits are shifted from the previous position. The tapped bits are XNORed with the feedback loop before a shift to the next position.

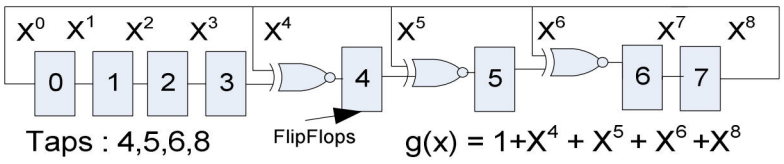


Figure 3-3: An 8-bit maximum-length LFSR.

The taps' combination and location are represented by a polynomial. Figure 3-3 shows an 8-bit maximum-length Galois LFSR, its taps and its polynomial.

By appropriately selecting the tap locations, it is possible to build a maximum-length LFSR of any size with either two or four taps [44] [6]. Additionally, when interconnect delay and feedback path's fan-out are ignored, the latency of the maximum-length LFSR will be independent of its width (size) [63] [6]. As Section 3.4.4 shows, latency increases only slightly with the LFSR size, primarily due to the increased capacitive load on the control lines.

3.2.2 Up/Down LFSRs

The tap locations for a maximum-length, unidirectional n-bit LFSR can be represented by a primitive polynomial g(x) as depicted in (7):

$$g(x) = \sum_{i=0}^{n} C_i \times X^i \qquad (C_0 = C_n = 1) \tag{7}$$

In (1), X^i corresponds to the output of the i-th bit of the shift register and the constants C_i are either 0 (no tap) or 1 (tap). Given g(x), a primitive polynomial h(x) for an LFSR generates the reverse sequence as depicted in (8) [10]:

$$h(x) = \sum_{i=0}^{n} C_i \times X^{n-i} \qquad (C_0 = C_n = 1) \tag{8}$$

The superposition of the original and reverse LFSRs forms a reversible "up/down" LFSR. The up/down LFSR consists of a shift register similar to the one used for the uni-directional LFSR; a 2-to-1 multiplexer per bit to control the shift direction; and, twice as many XNOR gates as the

Figure 3-5: A 3-bit maximum-length up/down LFSR.

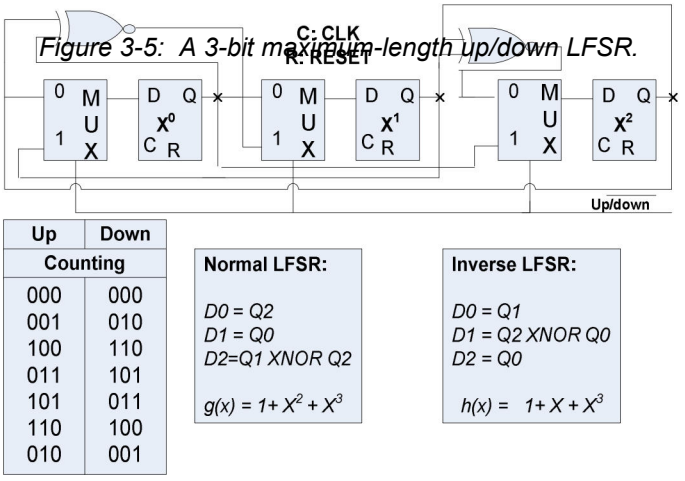


Figure 3-4: A 3-bit maximum-length up/down LFSR.

unidirectional LFSR. Figure 3-5 shows the construction of a 3-bit maximum-length up/down LFSR. It also depicts the polynomials and count sequence of both up and down directions. In general, it is possible to construct a maximum-length up/down LFSR of any width with two or six XNOR gates (i.e., four or eight taps) [6].

3.2.3 Comparison with Other Up/Down Counters

This section compares LFSR counters with other synchronous up/down counters that could be a choice of counter for CBFs. We restrict our discussion to synchronous up/down counters of width n with a count sequence of at least 2^n-1 states.

The simplest type of synchronous counter is the binary modulo-2^n n-bit counter. For this counter, speed and area are conflicting qualities due to carry propagation. For example, the n-bit ripple-carry synchronous counter, one of the simplest counters, has a latency of O(n) [63]. Counters with a Manchester carry-chain, carry-look-ahead, or binary-tree carry propagation have latencies of O(log n) at the cost of more energy and area [64]. In applications where the count sequence is

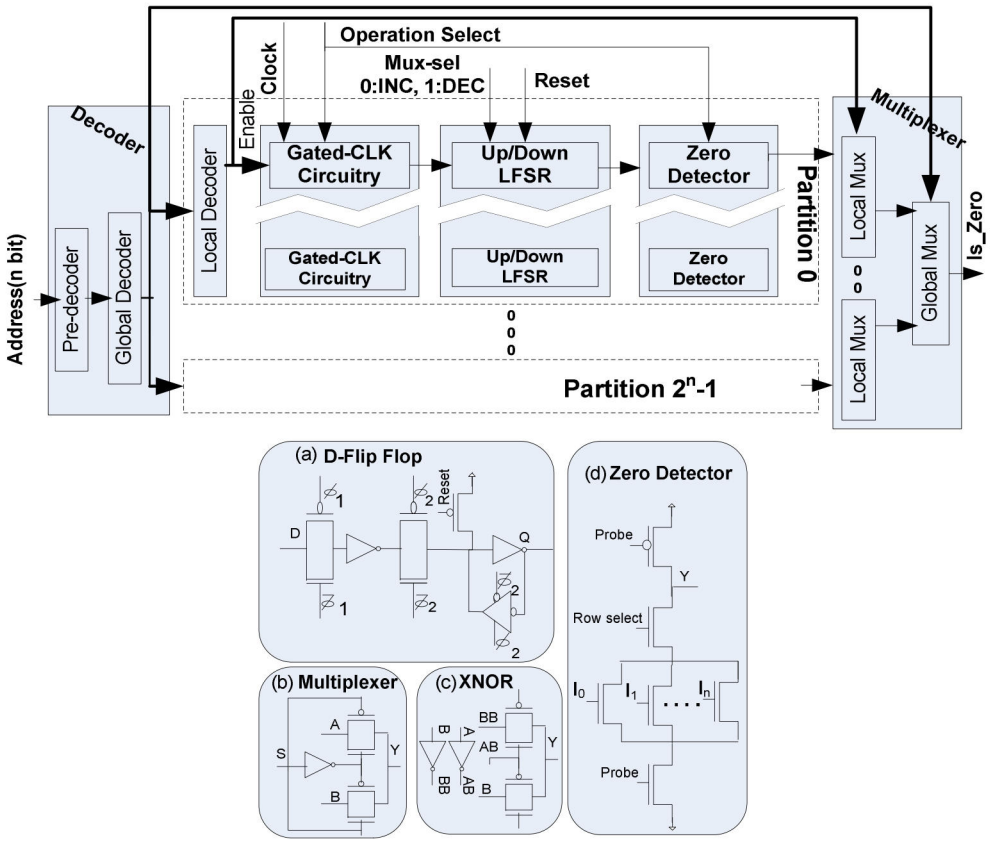


Figure 3-6: The architecture of L-CBF and the basic cells of an up/down LFSR: (a) two-phase flip-flop, (b) 2-to-1 multiplexer, (c) XNOR gate, and (d) embedded zero detector.

unimportant (e.g., circular FIFOs' pointers and frequency dividers), an LFSR counter offers a speed-, power- and area-efficient solution. The LFSR latency is nearly independent of the LFSR size. Specifically, the LFSR latency comprises the latencies of the flip-flop, the XNOR gate and the feedback loop. The feedback-loop latency is the propagation latency of the last flip-flop's output to the input of the farthest XNOR gate from this last flip-flop. Ignoring secondary effects on the feedback-path latency, the n-bit maximum-length LFSR's latency is O(1) and independent of the LFSR size [63] [6]. These characteristics make LFSRs a suitable counter choice for CBFs.

3.2.4 L-CBF Implementation Details

Figure 3-6 depicts the high-level organization of the L-CBF. L-CBF includes a hierarchical decoder and a hierarchical output multiplexer. The core of the L-CBF design is an array of up/down LFSRs and zero detectors. The L-CBF design is divided into several partitions where each row in a partition consists of an up/down LFSR and a zero-detector. The number of partitions is chosen as to reduce power and latency.

L-CBF accepts three inputs and produces a single-bit output *is-zero*. The input *operation select* specifies the operation type: INC, DEC, PROBE and IDLE. The input *address* specifies the address in question, and the input *reset* initializes all LFSRs to the zero state. The LFSRs utilize two non-overlapping phase clocks generated internally from an external clock. We use a hierarchical address decoder that minimizes the energy-latency product [7]. The decoder comprises a pre-decoder, a global decoder to select the appropriate partition, and a set of local decoders, one per partition. Each partition has a shared local is-zero output. A hierarchical multiplexer collects the local is-zero signals and provides the single-bit *is-zero* output.

Figure 3-6 also depicts the basic cells of each up/down LFSR and zero decoder. Shown are the flip-flop used in the shift registers, the multiplexer controlling the change direction ("up"/"down"), the XNOR gate and a bit-slice of the zero decoder. Further details on the L-CBF implementation are presented in Section 3.3.

3.2.4.1 Multi-Porting

Some applications require multiple, simultaneous CBF accesses. In the simplest implementation, the CBF can be banked to support simultaneous accesses to different banks. This mirrors the

organization of high-performance caches that are often banked to support multiple accesses instead of being truly multi-ported. However, not all possible combinations of the CBF accesses can be supported with this organization. If two accesses require accessing to the same bank, they have to be serialized. True multi-porting is straightforward by replicating in the case of simultaneous accesses to different counts. For S-CBF, we need an SRAM with multiple read and write ports and multiple shared up/down counters. For L-CBF, we need to replicate the decoder, the zero detectors and the output multiplexer.

In case of multiple accesses to the same count, multi-porting is not straightforward. A simple solution includes detecting, and then serializing these accesses. Alternatively, extra circuitry can be added to determine the collective effect of all accesses. For example, the net effect of two simultaneous increments is increasing the counter value by two. For S-CBF, this circuitry can be embedded into the shared counter. For L-CBF, the capability of shifting by multiple cells per cycle for each LFSR is required. Our work does not investigate these enhancements.

3.3 Analytical Model

Analytical models help computer architects estimate the latency and energy of various architectural alternatives under exploration without first developing a physical-level implementation. To the best of our knowledge, no such analytical models for CBFs exist. This section presents analytical models for the worst case latency and energy (dynamic and leakage) of the L-CBF implementation. These analytical models can be incorporated into architectural-level power-performance simulators such as Wattch [12].

The models predict L-CBF's latency and energy as a function of the entry count, the entry width and the number of banks. The models were developed based on our L-CBF's full-custom

Table I : Analytical Model Input Parameters

Externally Visible Organizational Parameters	
NoE	Number of entries
WoE	Width of each entry (Count width)
Internal Organizational Parameters	
NoRP	Number of rows per partition
Technology Parameters	
C_w, R_w	Per unit length capacitance and sheet resistance of metal layers
Other parameters as in [69], such as C_{gate} , $C_{ndiffarea}$, $C_{ndiffside}$, $C_{ndiffgate}$, $C_{pdiffarea}$, $C_{pdiffside}$, $C_{pdiffgate}$, $R_{eq.nmos}$, $R_{eq,pmos}$, V_{dd}.	

implementation in a 0.13 μm fabrication technology (Section 3.2.4).

In our implementation and models, the gates are sized to have equal rise and fall delays. The models do not account for the external loads as they are independent of the CBF implementation. While it is feasible to extend the models to predict the latency and energy for other fabrication technologies, this extension is left for future work. The 0.13 μm technology was the best technology available to us at the time of experimentation.

The rest of this section is organized as follows: Section 3.3.1 discusses the methodology that we used for developing the analytical models and the input parameters of the models. Section 3.3.2 and Section 3.3.3 present the latency and energy models respectively. Discussing the accuracy of the model estimations is postponed until Section 3.4.5 where we compare the model estimations against the simulation results.

3.3.1 Methodology

To model latency and energy per operation, we decompose the design into several RC circuits. Our analysis methodology is similar to that of CACTI [65] [69]. RC circuit analysis requires estimations of the gate capacitance (C_{gate}), the diffusion capacitance ($C_{diffusion}$), the overlap capacitance (C_{ovelap}), the equivalent on resistance for NMOS transistors ($R_{eq\text{-}nmos}$) and the equivalent on resistance for PMOS transistors ($R_{eq\text{-}pmos}$). Information such as transistor sizes and the interconnect lengths, required for capacitance and resistance estimations, is extracted from the full-custom layout. More details on the estimation of C_{gate} ,$C_{diffusion}$, C_{ovelap} , $R_{eq\text{-}nmos}$ and $R_{eq\text{-}pmos}$ can be found in [30] [48].

Transistors are scaled to minimize the energy-latency product for larger CBFs. Table I lists the model input parameters that fall under three broad classes: externally-visible organizational parameters, internal organizational parameters and technology-specific parameters. The externally-visible L-CBF organization is defined by the total number of entries, NoE, and the width of each entry count, WoE. Internally, L-CBF can be partitioned into the banks with NoRP rows in order to balance or improve power and latency.

3.3.2 Latency Model

This section presents an analytical model for the L-CBF's worst-case latency. Figure 3-7 through Figure 3-9 depict the equivalent RC circuit along the critical path. For clarity, a label is assigned to each element in the path. Subscripts specify the element's corresponding resistance and capacitance, the gate type (e.g., inverter) and the capacitor type (e.g., drain: d, source: s, and gate: g). The update operation's latency comprises the decoder latency, the row clock driver latency and the up/down LFSR latency. The probe operation's latency comprises the decoder latency, the zero-detector latency, and the output-multiplexer latency. The following subsections discuss the latency analysis for each component (e.g., decoder) focusing on resistance and capacitance estimation. Then, we present the analytical latency models of each CBF operations.

3.3.2.1 Component Delay: Decoder

Figure 3-7 (a) through (f) show the decoder's simplified critical path and its equivalent RC circuit. To estimate the RC latency, we determine the number and the size of the transistors and interconnects along the critical path. Almost all of them are a function of NoE and/or NoRP. The decoder utilizes a hierarchical architecture. In the pre-decode stage, each 3-to-8 decoder generates a 1-of-8 code for every three address bits. If the number of address bits is not divisible by three, a 2-to-4 decoder or an inverter is used. Each x-to-2^x decoder is implemented using 2^x NAND gates and x inverters to complement the address inputs. In the second stage, the pre-decoder outputs are combined using NOR gates. When beneficial, an inverter chain is used at the pre-decode stage's output to reduce latency. The decoder latency is the time it takes for an

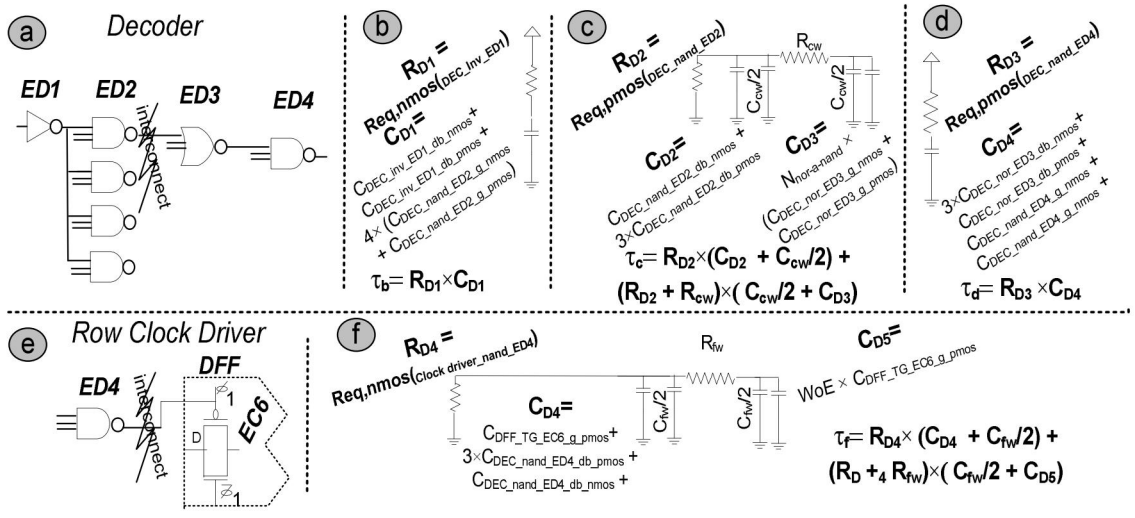


Figure 3-7: RC circuit analysis along the critical path of L-CBF (decoder and row clock driver).

address input that passes the threshold voltage of the inverter (ED1) to cause the output of the NOR (ED3) to reach the threshold voltage of the NAND (ED4). Equations (9) to (18) calculate subsequently the number of address bits (N_{addr}), the number of 3-to-8 decoders (N_{3to8}), the number of NOR gates (N_{nor}), and the fan-in of a NOR gate ($N_{nor\text{-}input}$) as a function of NoE. The formulas *Extra-2to4* and *Extra-inv* calculate whether an additional 2-to-4 decoder or an inverter is required when the number of address bits is not divisible by three. The formula $N_{nor\text{-}a\text{-}nand}$ calculates the number of NOR gates fed by a NAND gate. The distance between two subsequent NOR gates and the wire's equivalent resistance and capacitance are given by (16) and (17).

3.3.2.2 Component Latency: Row Clock Driver

Figure 3-7 (e) and (f) show the simplified critical path of the row clock driver and its equivalent RC circuit respectively. The NAND gate (ED4) performs clock gating. Its inputs are the global clock, decoder output and *operation select*. If a row is selected and the operation is an INC or DEC, the clock signal is applied to the addressed up/down LFSR. The worst case latency occurs when the clock signal is delivered to the last DFF. The wire length between the row clock driver and the last DFF (L_{fw}) is proportional to the LFSR width. This characteristic is also true for the length of the LFSR feedback path (L_{hw}). Both L_{fw} and L_{hw} are calculated by (18). This wire length is used to estimate equivalent resistance and capacitance.

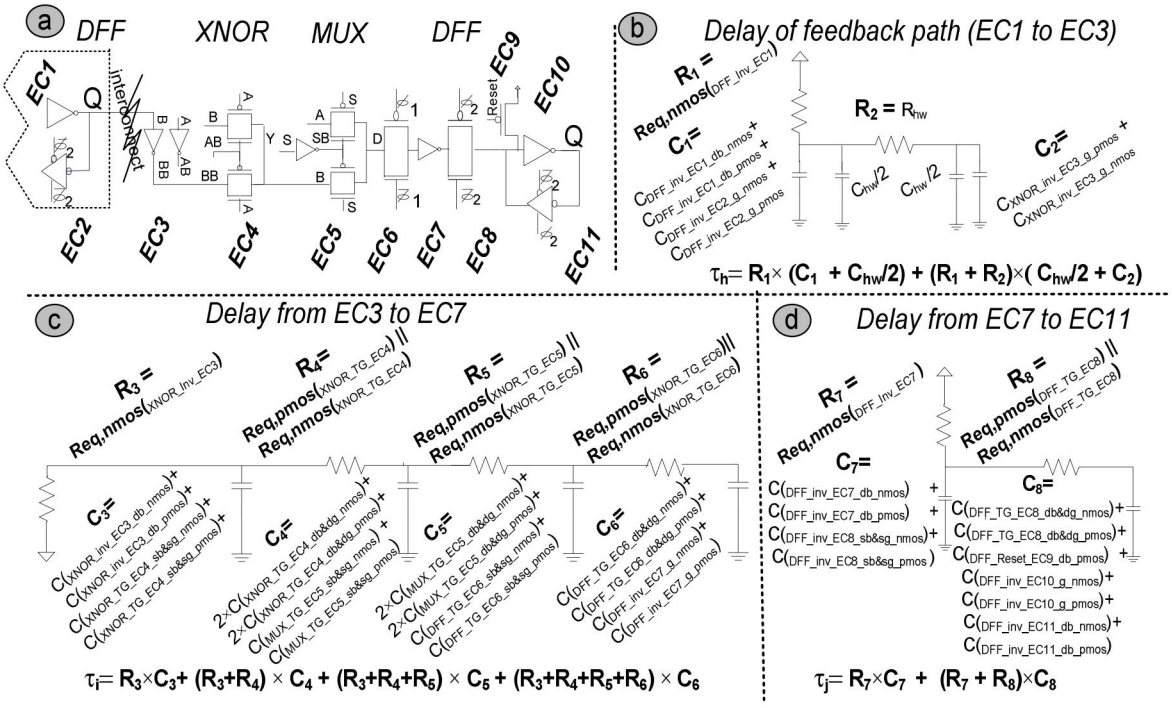


Figure 3-8: RC circuit analysis along the critical path of L-CBF (up/down LFSR).

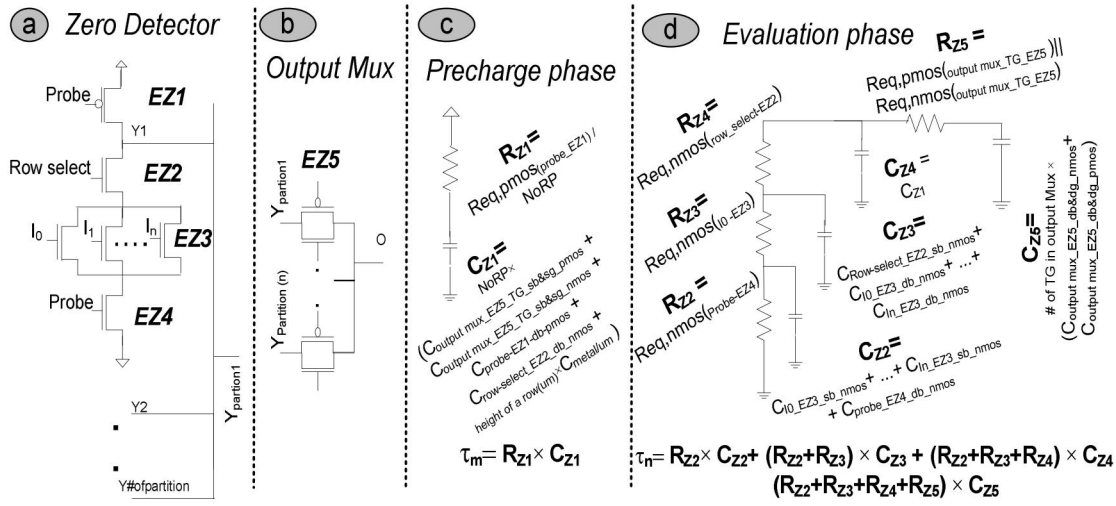


Figure 3-9: RC circuit analysis along the critical path of L-CBF (zero detector and output multiplexer).

3.3.2.3 Component Latency: Up/down LFSR

Figure 3-8 (a) through (d) show the equivalent RC circuit for the up/down LFSR. The feedback path's latency is the propagation latency of the last flip-flop's output to the input of the farthest XNOR gate. As addressed in Section 3.2.1, a maximum-length, n-bit up/down LFSR requires at most six XNORs [6]. The length of feedback path for a maximum-length, WoE-bit up/down LFSR is given by (20).

3.3.2.4 Operation Latency: Increment and Decrement

The update operation's latency comprises the decoder latency, the clock driver latency and the up/down LFSR latency. All the gates are sized to have the same rise and fall delays. The update operation's latency is calculated by (19), where τ_b through τ_j are time constants defined in Figure 3-7 and Figure 3-8 respectively.

3.3.2.5 Component Latency: Zero Detector and Output Multiplexer

The zero detectors of every set of NoRP rows in a partition have a shared output. This output is steered to the single-bit output, *is-zero*, through the output multiplexer. A probe operation involves three stages: *(i)* decode and precharge, *(ii)* evaluation, and *(iii)* transfer to the output. The decoding stage is the same for updates and probes. The pre-charging process is concurrent with the decoding. In the precharge stage, the shared output of a partition is charged to the supply voltage, V_{dd}. During the evaluation stage, based on the associated up/down LFSR's value, the

partition output is discharged to zero or stays at V_{dd}. The selected partition's output is transferred to the *is-zero* output by the output multiplexer.

$$N_{addr} = \lceil \log_2 (NoE) \rceil \quad (9)$$

$$N_{3to8} = \lfloor 1/3 \times (N_{addr}) \rfloor \quad (10)$$

$$Extra\text{-}2\text{-}to\text{-}4 = \lfloor 1/2 \times [N_{addr} - (3 \times N_{3to8}] \rfloor \quad (11)$$

$$Extra\text{-}inv = N_{addr} - (3 \times N_{3to8}) - (2 \times Extra\text{-}2\text{-}to\text{-}4) \quad (12)$$

$$N_{nor} = NoE \quad (13)$$

$$N_{nor\text{-}inputs} = (N_{3to8} + Extra\text{-}2\text{-}to\text{-}4 + Extra\text{-}inv) \quad (14)$$

$$N_{nor\text{-}a\text{-}nand} = NoE/8 \quad \text{if } (N_{addr} \text{ is divisible by 3}) \quad (15)$$

L_{cw} (μm) = wire length between two NOR gates fed by the same NAND(ED2) gate in the predecode stage.(extracted from the layout) (16)

$$R_{cw}(Ohm) = R_{Ohm/\square} \times (L_{wire}/W_{wire}), \quad (17)$$

$$C_{cw}(Farad) = C_{(Farad/um)} \times L_{wire}(\mu m)$$

$$L_{hw}(\mu m) = (\text{width of DFF} + \text{width of Mux}) \times (WoE - 6) + (\text{width of DFF} + \text{width of XNOR} + \text{width of MUX}) \times 6 \quad (18)$$

$$\text{Latency}_{Update} = 0.69 \times (\tau_b + \tau_c + \tau_d + \tau_f + \tau_h + \tau_i + \tau_j) \quad (19)$$

$$\text{Latency}_{Probe} = 0.69 \times (\tau_b + \tau_c + \tau_d + \tau_m + \tau_n) \quad (20)$$

3.3.2.6 Operation Latency: Probe

Figure 3-9 (a) through (d) depict the equivalent RC circuits for the zero detector and the output multiplexer. The probe operation's latency comprises the decoder latency, the zero detector latency and the output multiplexer latency. The latency is calculated by (20), where τ_b to τ_n are time constants that are presented in Figure 3-7 and Figure 3-9.

3.3.3 Energy Model

Four sources of power dissipation are as follows: First is the dynamic switching power due to the charging and discharging of the circuit capacitances. Second is the leakage power due to the reverse-biased diodes and sub-threshold conduction. Third is the short-circuit current power due to the finite signal rise/fall times. Fourth is the static biasing power found in some types of logic styles (e.g., pseudo-NMOS). For the given technology, circuit simulations suggest that the first two are the principal sources of energy consumption.

3.3.3.1 Dynamic Power

Dynamic power is the result of the gate output transitions. Output transitions cause the capacitive load driven by the gate to be charged or discharged. To approximately calculate the capacitive load required for the energy per operation estimation, the gate (e.g., NAND) and interconnect capacitances in the signal path are added up. The energy dissipated per transition (0-to-1 or 1-to-0) is given by (21) where C_L is the load capacitance, V_{dd} is the supply voltage, and ΔV is the output's voltage swing.

$$E_{dynamic} = 0.5 \times C_L \times V_{dd} \times \Delta V \quad (21)$$

The analytical energy models use the capacitance estimations of the RC delay analysis. For instance, the decoder energy is calculated by adding up the gate and interconnect capacitances along the critical path as given by (22). The same methodology is used for other components.

$$E_{decoder} = 0.5 \times V_{dd}^2 \times (C_{D1} + C_{D2} + C_{cw}+C_{D3}) \quad (22)$$

3.3.3.2 Leakage Power

This section discusses the leakage power calculation methodology. To calculate the leakage current in a MOSFET, similar to the work of Mamidipaka et al. [41], we use the model proposed by Zhang et al. [71] given by (23).

$$I_{lkg} = \mu_0 \times C_{ox} \times \frac{w}{l} \times e^{b(V_{dd}-V_{dd0})} \times v_t^2 \times (1 - e^{-V_{dd}/v_t}) \times e^{\frac{-v_{th}-v_{off}}{nv_t}} \quad (23)$$

For a given threshold voltage (V_{th}) and temperature (T), all terms except the width (W) are constant for all the transistors in a given fabrication technology [41] [18]. Hence, (23) can be reduced to (24), where I_l is the leakage for a unit-width transistor at a given T and V_{th}.

$$I_{lkg} = W \times I_l(T, V_{th}) \quad (24)$$

We identify the distribution of the inputs for each component (e.g., single transistors or gates) based on the L-CBF's operation characteristics. Then, we derive worst case $I_l(T,V_{th})$ for each component considering different input states. Finally, we sum all the $I_l(T,V_{th})$s for all components. When stacks of transistors (transistors connected in series drain to source) exist in a design, leakage current reduces significantly [41]. The leakage characteristics of NMOS and PMOS transistors can be different from each other in a given fabrication technology. We assume that the $I_{lP}(T,V_{th})$ and $I_{lN}(T,V_{th})$ for the given technology are available to the models.

As an example, we discuss the leakage current calculation for the decoder. In L-CBF, by activating the enable signal during the update and probe operations, the 3-to-8 pre-decoder outputs are triggered (stage one), and the output of one of the NOR gates will set to logic value one (stage two). Equations (25) and (26) modeled the worst-case leakage current of these stages.

$$I_{stage\ 1} = N_{3to8} \times (3 \times I_{ED1} + 8 \times I_{ED2}) \quad (25)$$

$$I_{stage\ 2} = NoE \times (I_{ED3}) \quad (26)$$

$$I_{dec} = I_{stage1} + I_{stage2} \quad (27)$$

The leakage current for the decoder is given by (27). As another example, we discuss how the memory core's leakage current for the idle (or precharge) state is calculated. This memory core is assumed to comprise single-port SRAM cells similar in structure to the cell depicted in Figure 2-2. During the idle time, all wordlines are inactive, and BLs and BLBs are pre-charged to V_{dd}. We identify the off transistors during idle time and add up their leakage current as calculated in (28) and (29). Multiplying the I_{dec} by V_{dd} gives the leakage power estimation.

$$I_{memCellIdle} = I_{leakage}(N1) + I_{leakage}(N3) + I_{leakage}(p2) = (W_{N1} + W_{N3}) \times I_{lN} + WP2 \times I_{lP} \quad (28)$$

$$I_{memCoreIdle} = N_{rows} \times N_{cols} \times [(W_{N1} + W_{N3}) \times I_{lN} + WP2 \times I_{lP}] \quad (29)$$

3.4 Experimental Results

This section compares the energy, latency and area of S-CBF and L-CBF implementations. Moreover, for L-CBF implementation, this section compares the analytical model estimations against simulation results.

We compare S-CBF and L-CBF on a per operation basis. Both designs are implemented using the Cadence(R) tool set in a commercial 0.13µm fabrication technology. We developed transistor-level implementations and full-custom layouts for both designs optimized for the energy-delay product. We employed Spectre™ for circuit simulations. Spectre™ is a vendor-recommended simulator for design validation prior to manufacturing.

The rest of this section is organized as follow. We initially consider a 1K-entry CBF with 15-bit counts as it is representative of the CBFs used in previous proposals [46] [54]. Then, we present

Table II : Energy, latency and area of S-CBF and L-CBF implementations for a 1K-entry, 15-bit CBF.

	OPERATION	L-CBF	S-CBF	S-CBF/ L-CBF
Latency (ps)	INC/DEC	447.26	1670	3.7
	PROBE	580.32	910.12	1.6
Energy (pj)	INC/DEC	38.73	88.98	2.3
	PROBE	30.36	41.02	1.4
Area (mm^2)		0.95	0.30	0.31

results for other CBF configurations. Section 3.4.1 compares the energy, latency and area of L-CBF and S-CBF for updates and probes. Section 3.4.4 studies how energy and latency change as CBF's entry count and count width vary. Section 3.4.5 discusses the models' accuracy by comparing the model estimations against the simulation results.

3.4.1 Latency and Energy per Operation

We compare implementations of a 1K-entry, 15-bit count per entry CBF. For S-CBF, an SRAM with a total capacity of 15Kbits is used. The SRAM is partitioned to minimize the energy-latency product. For S-CBF, we do not consider the latency and energy overhead of the shared counter since our goal is to demonstrate that L-CBF consumes less energy and it is faster. To further reduce energy for probe operations in S-CBF, we introduce an extra bit per entry; this bit is updated only when the count changes from or to zero as described in Section 3.1.5 (Z-bits). For probe, we only read this bit. Furthermore, we apply a number of latency and energy optimizations on S-CBF [7] [42]. In detail, we implement the divided word line (DWL) technique by adopting a two-stage hierarchical row decoder structure to improve speed and power [8] [42]. Moreover, we reduce power further via pulse operation techniques for the wordlines, the periphery circuits and the sense amplifiers [42]. We also use multi-stage static CMOS decoding [7] and current-mode read and write operations to further reduce power [42]. For L-CBF, we utilize 16-bit LFSRs such that the LFSR can count at least 2^{15} values.

Table II shows the latency in picoseconds, the energy (static and dynamic) per operation in pico-joules, and the area in square millimeters for L-CBF and S-CBF. The last column reports the ratio of S-CBF over L-CBF per metric. The two rows per category report measurements for the update and probe operations respectively. For latency and energy, we report the worst case

values measured by appropriately selecting inputs. The latency and energy of the shared counter for S-CBF is not included; otherwise, the actual latency and energy of S-CBF would be higher. As observed from Table II, L-CBF is 3.7 and 1.6 times faster than S-CBF during update and probe operations respectively. In addition, L-CBF consumes 2.3 or 1.4 times less energy than S-CBF for update and probe operations respectively. These significant gains in speed and energy consumption come at the expense of more area. L-CBF requires about 3.2 times more area than S-CBF. However, as discussed in Section 3.2, area is less of a concern in modern microprocessor designs due to the abundance of resources.

Disregarding the latency and energy overhead of the shared counter, the measurements for the S-CBF are optimistic. An up/down 15-bit LFSR counter has a latency of 240 ps and energy per update operation of 25 fj. If this LFSR was used as the shared counter for the S-CBF, L-CBF would be 4.3 or 1.98 times faster than S-CBF for updates and probes respectively (relative energy remains virtually the same).

3.4.2 Per Component Energy Breakdown

Figure 3-10 shows a per component breakdown of total energy consumption for both S-CBF and L-CBF. Depicted in Figure 3-10, most of the energy (79% and 74% respectively for updates and probes) in S-CBF is consumed by the memory core (world-lines, bitlines and SRAM cells). The decoder and sense-amplifiers consume considerably less energy. This energy distribution is expected as we applied aggressive energy and latency optimizations to these components. Although the distribution of static and dynamic energy values has not been shown in figures, it should be noted that for L-CBF probes, about 50% of the total energy is dissipated in inactive components such as the LFSR array and row drivers. For L-CBF updates, 50% of the total energy is dissipated in non-selected LFSRs, non-selected row drivers, inactive zero detectors and inactive output multiplexer. In brief, about 50% of the total energy is dissipated in active components during update and probe operations.

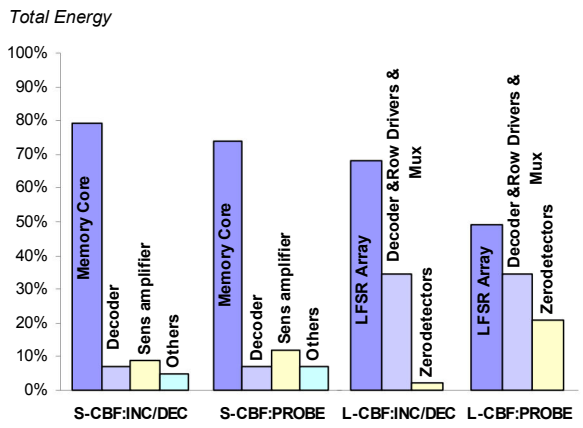


Figure 3-10: Per component energy breakdown for S-CBF and L-CBF ((INC/DEC) and (PROBE)).

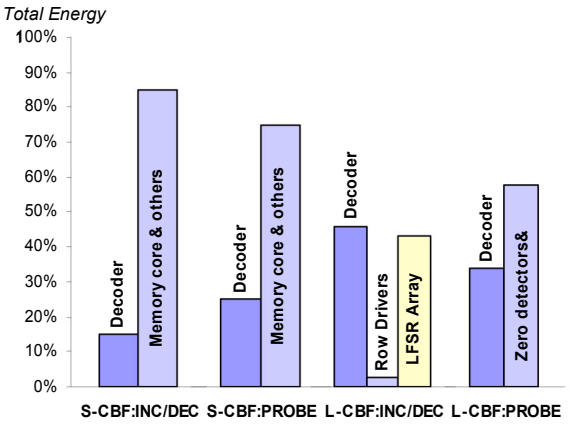


Figure 3-11: Per component latency breakdown for S-CBF and L-CBF ((INC/DEC) and (PROBE)).

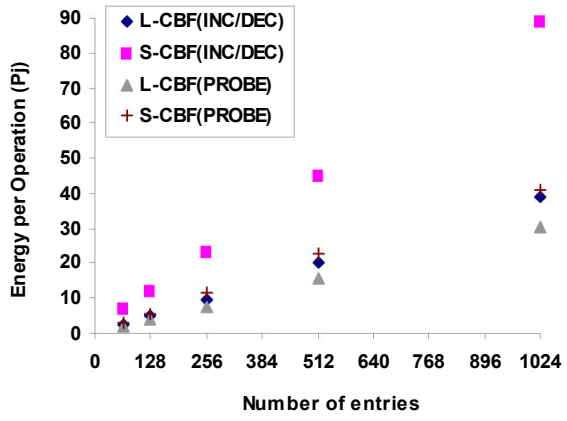


Figure 3-12: Energy per operation as a function of the number of entries for L-CBF and S-CBF with count width of 15-bit.

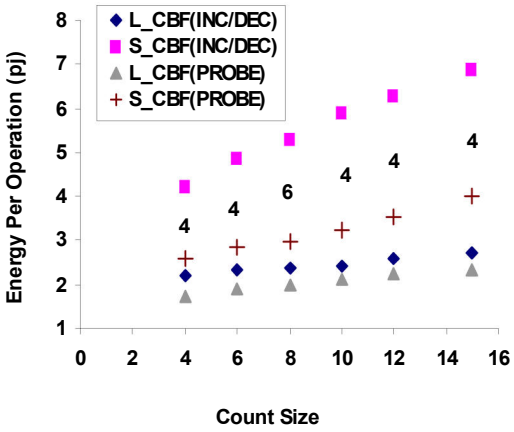


Figure 3-13: Energy per operation as a function of the count width for L-CBF and S-CBF for a 64-entry CBF.

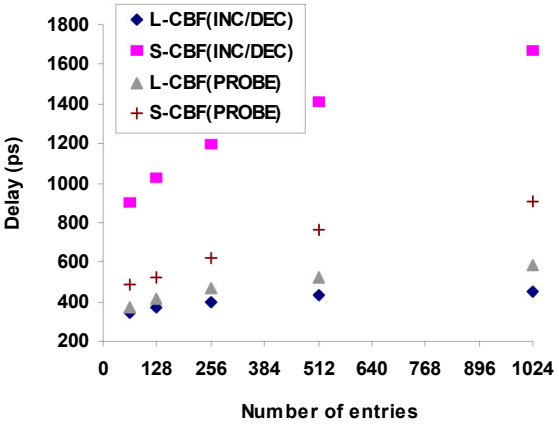


Figure 3-14: Latency as a function of entry count for L-CBF and S-CBF with15-bit counts.

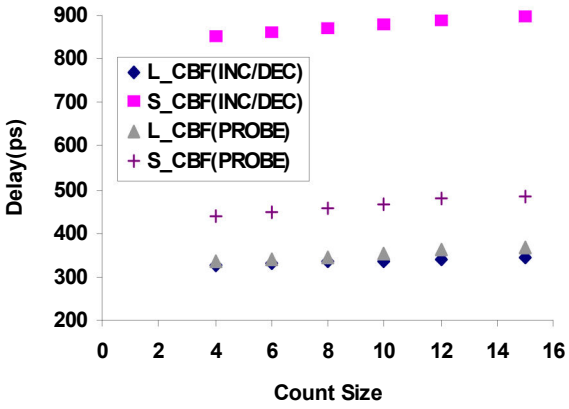


Figure 3-15: Latency as a function of count width for L-CBF and S-CBF with 64 entries.

3.4.3 Per Component Latency Breakdown

Figure 3-11 shows a per component breakdown of latency for both S-CBF and L-CBF for updates and probes. In S-CBF, the update operation latency comprises the decoder latency, the SRAM read access latency (excluding decoder latency) and the SRAM write access latency (excluding the decoder latency). In detail, the update operation latency comprises the decoder latency, the read-wordline latency, the read-bitline latency, the read-sense amplifier latency, the read-output multiplexer latency, the write-write driver latency, the write-wordline latency, the write-bitline latency and the precharge latency. The precharge latency is included since the update operation involves a read-modify-write sequence. In S-CBF, significant part of the latency belongs to the memory core, demonstrating that significant potential exists for improvements with L-CBF. For L-CBF, the update operation's latency comprises the decoder latency, the row clock driver latency and the up/down LFSR latency. For L-CBF, the probe operation latency comprises the decoder latency, the zero detector latency and the output multiplexer latency. In L-CBF, the latency is balanced across the LFSR core and the decoder demonstrating that the L-CBF successfully reduces latency compared to S-CBF.

3.4.4 Sensitivity Analysis

This section investigates latency and energy variations as a function of the number of entries and the width of each entry count (count width) for both L-CBF and S-CBF.

3.4.4.1 Energy per operation

Figure 3-12 reports the CBF's energy per operation as a function of entry count for 64- through 1K-entry CBFs in power of two steps. L-CBF consistently consumes less energy than S-CBF and the relative difference increases slightly for larger entry counts.

Figure 3-13 reports the energy per operation as a function of count width in the range of four to 16 bits for a 64-entry CBF. Along L-CBF measurements, we also report the number of taps needed by each count width (either four or eight). We observe that the L-CBF's energy scales better than S-CBF's energy. Communication in L-CBF is local and primarily between adjacent storage cells. For this reason, increasing the number of cells does not impact the overall energy significantly. The energy of the S-CBF increases at a greater rate because additional bitlines and

sense amplifiers are introduced and the wordlines become longer. Figure 3-13 shows that changing the number of LFSR taps from four to eight does not significantly impact the energy.

3.4.4.2 Latency

Figure 3-14 reports the latencies for CBFs of 64 through 1K entries in power of two steps. As the number of entries increases, the size and the latency of the decoder increase and so does the size and latency of the output multiplexer. L-CBF is consistently faster than S-CBF. The difference in speed increases slightly with the number of entries.

Figure 3-15 reports the latency as a function of the LFSR width in the range of four to 16 bits for a 64-entry CBF. We observe a negligible increase in the latency of update operation as the width increases. For larger LFSR widths, three potential sources of increased latency are as follows: the row clock driver, the LFSR feedback loop and the embedded zero detector. Increasing the LFSR width increases the load on the per row clock driver. By resizing the row driver or by adding a buffer chain, any significant latency increase can be avoided at the cost of more energy. As the counter width increases, so does the feedback loop's length, and hence the LFSR's latency. As discussed earlier, in practice, this increase is negligible for the widths considered in this study. Increasing the LFSR width increases the number of zero detector's inputs and hence its latency. We observe that L-CBF's latency increases slightly for wider counts compared to S-CBF's.

3.4.5 On the Accuracy of the Analytical Models

This section discusses the accuracy of the analytical models. In this analysis, the relative estimation error is calculated by (30):

$$\%Error = \frac{Analytical - Simulation}{Simulation} \times 100 \quad (30)$$

Figure 3-16 and Figure 3-17 compare circuit measurements with analytical model estimations for energy and latency as a function of L-CBF's entry count. The circuit measurements are reproduced from Figure 3-12 and Figure 3-14 respectively. The worst-case relative error per operation is also depicted. The worst-case relative errors for the energy and latency are respectively within 10% and 5% of the Spectre™ circuit simulation results. As observed, the

error is monotonic and the estimations are in agreement with the simulation results in predicting the trend of latency and energy per operation variations.

Analytical model estimations may differ from simulation results due to several factors. Comparisons of the model-estimated and layout-extracted capacitances demonstrate that about 5% of the error is due to capacitance estimation inaccuracy. The formulas used to calculate gate and diffusion capacitances are over-simplified, and the capacitances are assumed to be voltage independent. The energy model exhibits a worst-case error of about 10%. The leakage power model accounts for 4.5% of this error. Leakage current largely depends on the circuit state and, hence, without circuit simulations, the leakage power cannot be accurately quantified.

3.4.6 Simplified Latency and Energy Models

The models discussed in Section 3.3 are quite detailed and require input parameters derived from the physical-level implementation and target fabrication technology. Often, the physical-level implementation details are unavailable, and getting reasonable estimates could be difficult. Specifically, often processor designs start several years (five is typical) before the fabrication. At the early stages of the design process, the physical-level implementation details and fabrication technology specific parameters are unknown as the technology itself may be under development. Accordingly, this section, presents simplified models, which provide rough estimates of relevant

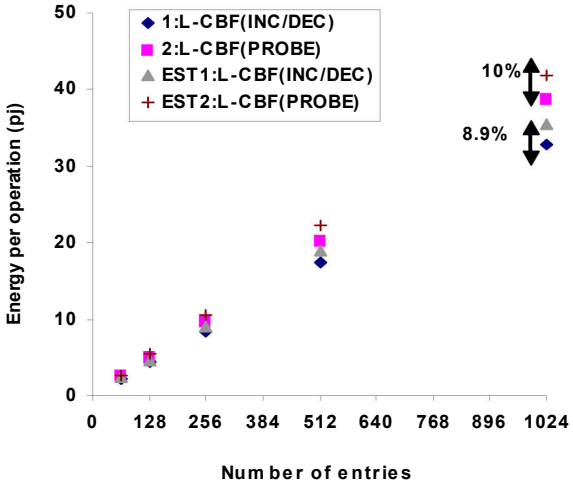


Figure 3-16: Energy per operation as a function of entry count for L-CBF with 15-bit counts

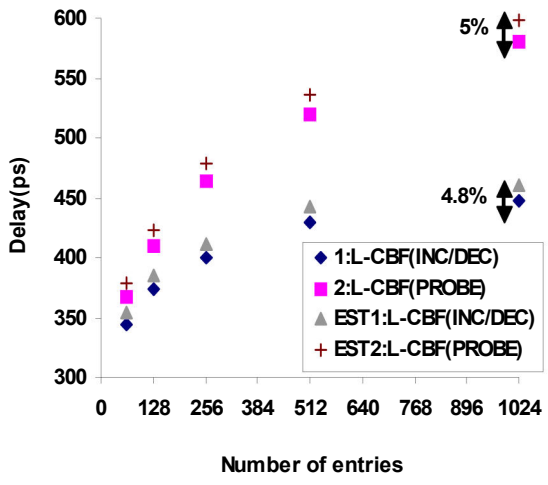


Figure 3-17: Latency as a function of number of entries for L-CBF with 15-bit counts

Spectre™ simulation results and analytical model estimations.

characteristics. The empirical models are derived from measurements over several actual designs. In particular, we applied curve fitting over 22 data points, based on the measurements from our physical-level implementations in a 130 nm technology [8]. The result of curve fitting is a function that closely fits the data points. Using such a function, computer architects can follow the variation trend of the data set and analyze quantitatively the underlying data for forecast purposes through interpolation and extrapolation. In numerical analysis, interpolation is a method of constructing new data points within the range of a discrete set of known data points (interpolation is a specific case of curve fitting). Extrapolation is the process of constructing new data points outside a discrete set of known data points. However, the extrapolation results are often subject to greater uncertainty, and the reliability of the results require a prior knowledge of the process used to obtain the data points.

Simplified latency and energy models are given by (31)-(34). WoE and NoE are model's input parameters, which are parameters that a computer architect may change during architectural-level exploration. Using our physical-level measurements, we observe that for a fixed NoE, increasing the WoE increases L-CBF's delay and energy logarithmically. We further observed that for a fixed WoE, increasing NoE increases L-CBF's delay and energy logarithmically. When compared with the actual circuit measurements (data points), the worst-case error given by

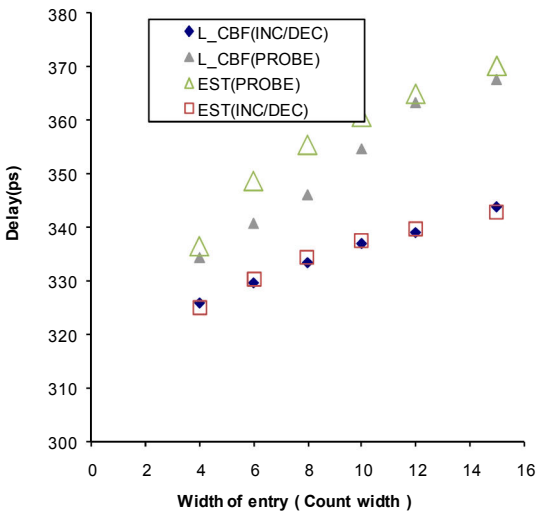


Figure 3-18: Latency as a function of entry count for L-CBF with 64-entries

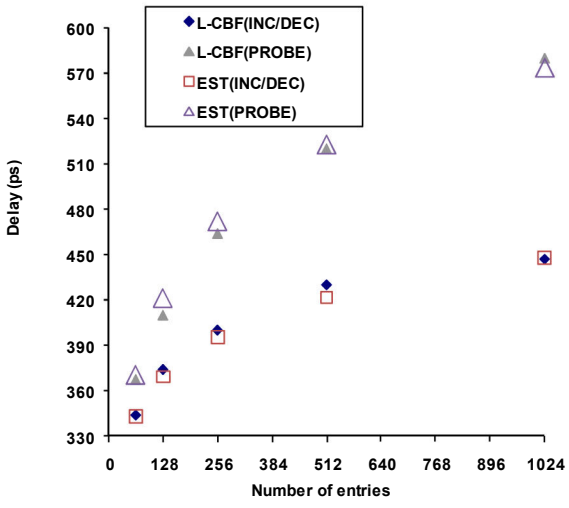


Figure 3-19: Latency as a function of number of entries for L-CBF with 15-bit counts

Spectre™ simulation results and simplified model estimations.

the curve is 12% and the average error is 4.5%. Figure 3-18 and Figure 3-19 show the data points (delay measurements from circuit implementation) and their corresponding fitted curves.

$$Delay_{L\text{-}CBF(INC/DEC)} = 13.465 \times Ln(WoE) + 37.99 \times Ln(NoE) + 148.02 \quad (31)$$

$$Delay_{L\text{-}CBF(PROBE)} = 23.5 \times Ln(WoE) + 73.1 \times Ln(NoE) \quad (32)$$

$$Energy_{L\text{-}CBF(INC/DEC)} = 2.0505 \times e^{0.0184 \times WoE} \times NoE^{0.9847} \quad (33)$$

$$Energy_{L\text{-}CBF(PROBE)} = 1.5766 \times e^{0.028 \times WoE} \times NoE^{0.9848} \quad (34)$$

The range of WoE and NoE covered by the data points are as follows: 0<=WoE<=15 and 0<=NoE<=1024. Beyond this range, the results are subject to some uncertainty since parts of the circuit may scale differently. For instance, for a block with more than 1024 entries, there are many different ways of banking the structure, and each one would result in possibly different latencies. The simplified models do not capture this effect.

3.5 Conclusion

This work investigates physical-level implementations of CBFs and proposes L-CBF; L-CBF is a novel implementation consisting of an array of up/down LFSRs and zero detectors. This work also compares L-CBF and S-CBF in terms of speed, energy consumption and area; S-CBF is the previously-assumed implementation consisting of an SRAM array of counts and a shared counter. This work has evaluated the energy, latency and area of L-CBF and S-CBF in a commercial fabrication technology. L-CBF is superior to S-CBF in both latency and energy at the expense of more area. Circuit simulations show that for a 1K-entry CBF with a 15-bit count per entry, L-CBF compared to S-CBF is 3.7x or 1.6x faster and requires 2.3x or 1.4x less energy depending on the operation. Additionally, this work presents analytical and empirical latency and energy models for the L-CBF implementation. These models facilitate the estimation of CBF's latency and energy during architectural-level exploration when developing a physical-level implementation is unavailable, undesirable or unaffordable due to the time/cost constraints. Comparisons demonstrate that the estimations provided by the analytical models are 10% and 5% of the Spectre™ circuit simulation results for energy and latency respectively.

4. Checkpointed, Superscalar Register Alias Tables

Register renaming eliminates false data dependencies and increases ILP. The register alias table (RAT), the core of register renaming, is a performance-critical component of modern, dynamically-scheduled processors. The RAT is critical from performance perspective because it is read and updated by almost all instructions in order as they are decoded. Hence, it must operate at the processor's clock frequency or it must be pipelined.

RAT complexity and size, and hence latency and energy depend on several architectural parameters. The wider the issues width (IW), the more ports the RAT will need. Further, as the window size (WS) increases, so does the number of physical registers, and hence the RAT's entry width or entry count for SRAM- or CAM-based RAT implementations respectively. RAT complexity and size also increase with the number of simultaneous threads supported by the processor. RAT complexity is increased further by the use of speculation, control-flow or otherwise. Speculative execution sends the processor down the most likely path of a conditional branch, potentially executing instructions that may have to be discarded if the wrong path was taken. On mispeculations, all incorrect instructions are squashed, and execution resumes by fetching instructions from the correct control path. On mispeculations, the processor state (including the RAT content) must be restored such that it does not contain any of the mappings introduced by the incorrectly-speculated instructions. Accordingly, modern RAT designs incorporate a set of global checkpoints (GCs), each contains a complete snapshot of all relevant processor state including the RAT. GCs are used to recover from mispeculations, control-flow or software or hardware interrupts (e.g., a page fault or a timer-induced context switch). Recovery using a GC is "instantaneous", i.e., it requires a fixed, low latency. As detailed later in this chapter, introducing GCs impacts RAT latency and energy.

In early dynamically-scheduled processors, such as the MIPS R10000, GCs were allocated to every speculated branch [70]. This policy was feasible because very few GCs were sufficient to achieve high performance. Modern processors, however, use much larger instruction windows (e.g., 128 vs. 32), and hence require considerably more GCs to maintain high performance. Accordingly, recent work have assumed that the policy of allocating a GC to every speculated

branch is impractical for modern processors [3] [4] [20] [46]. These studies developed GC count reduction techniques focusing on IPC performance to compare alternatives. However, IPC does not predict performance of the techniques that impact the clock period. Determining the actual relation between GC count and performance is imperative for understanding whether existing state-of-the-art RAT checkpointing solutions work sufficiently well, or whether further innovations are required. Our work improves upon previous RAT checkpointing work by investigating how increasing the GC count affects the RAT latency, and thus actual performance. Specifically, this work studies how RAT latency and energy vary as a function of the number of GCs, IW and WS using full-custom implementations in a 130 nm fabrication technology.

Unlike previous work that primarily focused on architectural-level evaluation, this work relies on both physical-level and architectural-level evaluations to study the actual performance and energy impact of GCs. For the architectural-level evaluation, both conventional and state-of-the-art confidence-based methods for selectively allocating GCs are considered [5]. For SRAM-based RAT (sRAT), this work shows that ignoring actual delay incorrectly predicts performance: In particular, performance does not monotonically increase with the number of GCs as IPC measurements suggest. Specifically, two components determine actual performance: First, with more GCs, fewer cycles are spent recovering from mispeculations, hence improving performance. Second, introducing more GCs increases sRAT latency and consequently increases the clock period and decreases overall performance. In most cases, using very few GCs (e.g., four) leads to optimal performance.

Previous work relied on register files' analytical models, which were not adjusted to appropriately model the checkpointed RATs. To facilitate further RAT checkpointing studies, this work presents analytical models for the latency and energy of the checkpointed sRAT. The model estimations are within 6.4% and 11.6% of Spectre™ circuit-simulation results for latency and energy respectively. This range of accuracy for model estimations is acceptable for architectural-level studies.

RAT implementations can be based on SRAM or CAM structures; we refer to these implementations as sRAT and cRAT respectively. For most of the aforementioned experiments,

we have focused on the sRAT implementation. However, this work also compares the latency and energy variation trends of cRAT with those of sRAT. We observed that the two implementation styles exhibit different scalability trends. Typically, sRAT is faster, more energy-efficient and less sensitive to the window size or issue width. The difference between sRAT and cRAT grows larger with increasing NoGCs. For example, 4-way sRAT lookups require 33% less energy and are 12.1% faster than cRAT lookups when the number of architectural and physical registers are 64 and 128 respectively. The differences grow to 416% for energy and 34.6% for latency when the number of physical registers increases to 512. cRAT is less sensitive to increasing the number of GCs. When the number of GCs passes a limit, cRAT becomes faster than its equivalent sRAT. For instance, with 64 architectural registers and 128 physical registers, having more than 20 GCs makes a 6-bit, 128-entry cRAT faster than its equivalent 7-bit, 64-entry sRAT.

Motivated by the aforementioned results and by the existence of commercial designs that use a large number of GCs (e.g., the latest PowerPC processor), we propose an energy optimization for cRAT where only the entries containing the latest mappings for architectural registers remain active during lookups. The energy savings are, for the most part, a function of the number of physical registers. For instance, for a cRAT with 128 entries, energy is reduced by 40%. sRAT is more scalable and energy-efficient than cRAT as WS and IW increase; hence, sRAT has been a preferable choice in most recent processors.

In summary, this work makes the following contributions: *(i)* It presents full-custom implementations for checkpointed RATs of dynamically-scheduled, superscalar processors in a 130 nm CMOS technology. Two checkpointing organization mechanisms, which differ in the way GCs are organized, allocated and de-allocated, have been considered; *(ii)* For all RAT operations, this work quantitatively determines sRAT's and cRAT's latency and energy as a function of the number of GCs, IW and WS. *(iii)* Using architectural-level simulations, this work estimates how performance is affected by sRAT latency for two different checkpointed sRAT implementations considering a state-of-the-art selective GC allocation policy [5]; *(iv)* It presents analytical models for the sRAT's latency and energy and compares the model estimations against physical-level simulation results; *(v)* It compares the energy and latency variation trends of sRAT

and cRAT implementations; and *(vi)* It proposes an energy optimization for cRATs.

The rest of this chapter is organized as follows: Section 4.1 reviews the RAT's role in modern processors, provides background on RAT implementations and RAT checkpointing, and discusses related work. Section 4.2 discusses two checkpointed RAT implementations and sRAT physical-level implementation. Section 4.2.3 discusses the cRAT physical-level implementation. Section 4.3 presents the latency and energy analytical models for the sRAT. Section 4.4 presents the results of the physical- and architecture-level evaluations for sRAT and cRAT. This section also compares the circuit simulation results against the model estimations for sRAT. Furthermore, this section compares the latency and energy variations of sRAT and cRAT. Finally, Section 4.5 summarizes our findings.

4.1 Background

This section provides an overview on register renaming, RAT implementations and RAT checkpointing. This section also reviews related work on the RAT implementations and RAT checkpoint reduction.

4.1.1 The Role of the RAT in Register Renaming

The register renaming logic maps the architectural register names used by the instructions into the physical registers implemented in the processor. Register renaming assigns a different physical register for each write to the same architectural register. As a result, this mapping removes false name dependences (write-after-write, WAW, and write-after-read, WAR) that artificially limit ILP. The number of physical registers is larger than the number of architectural registers. Physical register names are recycled when their values are no longer needed (i.e., all instructions that might consume the values have executed). For each architectural destination register, renaming logic allocates a physical register and records this mapping in the RAT so that the subsequent architectural source registers will correctly reference the physical registers holding their latest value. Figure 4-1 illustrates how false data dependencies are removed by register renaming for the architectural register R2.

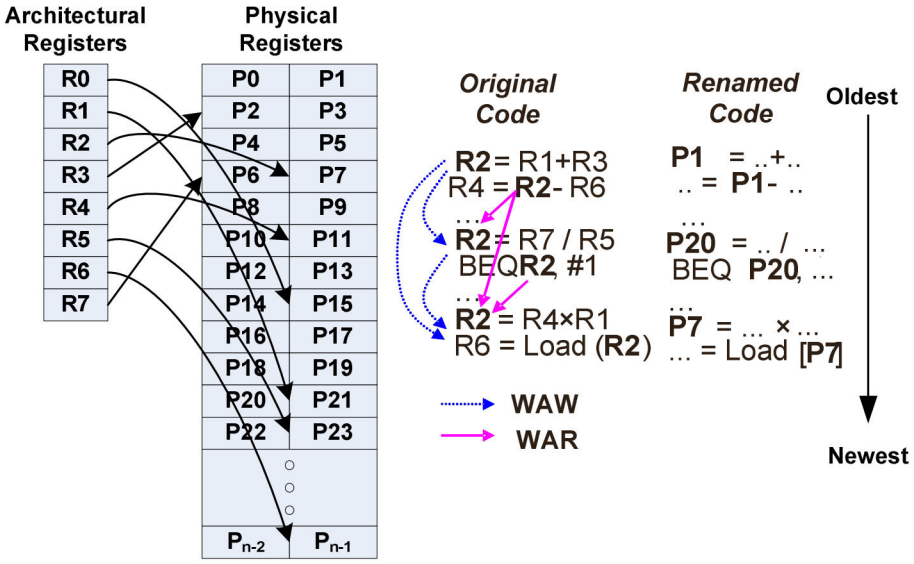


Figure 4-1: An Example of Register Renaming

Renaming a single instruction through the RAT proceeds as follows: *(i)* Reading the current mappings (physical register names) for the source registers; *(ii)* Reading the current mapping of the destination register. This old mapping is saved in the reorder buffer (ROB) to support speculative execution (detailed in Section 4.1.4); *(iii)* Acquiring a physical register name from the pool of free registers and update the RAT to establish the destination register's new mapping.

4.1.2 RAT Implementations

The two commonly-used RAT implementations are based on SRAM or CAM structures. The SRAM-based RAT (sRAT), depicted in Figure 4-2(a), is an SRAM array that has as many entries as the number of architectural registers. sRAT is similar in structure to a multi-ported register file. The physical register name/address for an architectural register name/address is read or updated via a direct access to the corresponding RAT entry. The RAT entry width is equal to that

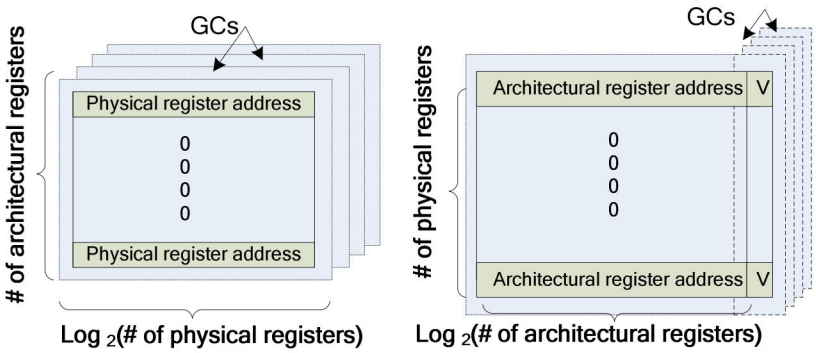


Figure 4-2: (a) SRAM-based RAT, (b) CAM-based RAT

of the physical register name (e.g., 7 bits for 128 physical registers). A GC contains the equivalent of a complete snapshot of the RAT. Mappings are checkpointed by copying the whole RAT content to a shadow table containing GCs. A GC contains as many bits as the RAT itself. MIPS R10000 [70] is an example of a processor that uses a checkpointed sRAT with four GCs.

The CAM-based RAT (cRAT) has as many entries as the number of physical registers; each RAT entry stores the architectural register name assigned to a given physical register in addition to a valid bit indicating whether this RAT entry corresponds to the most recent instance of the architectural register [49]. In this implementation (e.g., used in the Alpha 21264 [33], PowerPC [17]), a RAT lookup involves an associative search on the CAM content using the architectural register name as the key. A matching entry outputs the corresponding physical register address. In cRAT, depicted in Figure 4-2(b), the number of RAT entries is equal to the number of physical registers [33]. Each entry keeps the corresponding architectural register name and a valid bit. At any given point of time, only one entry per architectural register has the valid bit set (or *on*). When a new physical register is allocated, the architectural register name is written into the associated entry and the valid bit of the old mapping is cleared. For cRAT's GC, only the valid bits are copied to a shadow table. The Alpha 21264 [33] , for instance, uses an 80-entry cRAT with 80 GCs, which provides the capability of recovering the state associated with any of the 80 in-flight instructions. The total number of bits required by cRAT's GCs is 80 × 80, or 1.6K bits. By comparison, an equivalent sRAT with 80 GCs would require 64 × Log_2 (80) × 80, or 35.8K bits for the GCs given 80 physical registers and 64 architectural registers.

4.1.3 RAT Port Requirements

This work assumes a MIPS-like instruction set architecture (ISA), where the instructions may

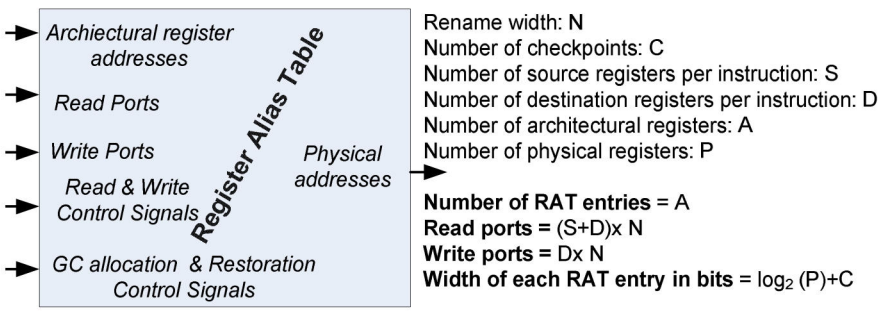


Figure 4-3: RAT Characteristics

have at most two source registers and one destination register. Given this assumption, the RAT needs to support 3xN reads and N writes per cycle, where N is the number of instructions required to be renamed per cycle. 2xN read ports are used to rename the two source operands, and another N read ports are needed to read the current mappings of the destination operands for the purpose of recovery using ROB (Section 4.1.4). Finally, the N write ports are used to write new mappings for the destination registers. Figure 4-3 shows a high-level block diagram of the RAT including its inputs and outputs.

4.1.4 Recovery Mechanisms

Modern processors utilize control-flow speculation to improve performance. When speculation is incorrect, all instructions along the mispeculated path must be squashed, i.e., any changes made by these instructions must be reversed. The two commonly used mechanisms for recovering the RAT content are the re-order buffer (ROB) and checkpointing. The ROB is a circular queue in which each instruction allocates an entry in program order during decode and releases the entry upon commit. Each ROB entry contains sufficient information for reversing the effects of the instructions along the mispeculated path. The ROB is a fine-grain, fail-safe recovery mechanism that can be used to recover from any exception in addition to branch mispeculations. To restore the RAT to the state it had at a particular instruction, all subsequent changes must be undone by traversing the ROB log in reverse order while writing back into the RAT the previous physical register names for the instructions' destination register. Accordingly, in ROB-based recovery, reversing the effects of each mispeculated instruction requires time proportional to the number of squashed instructions (typically, up to IW instructions may be squashed per cycle if the machine is capable of decoding IW instructions per cycle).

With the GC-based recovery mechanism, processors incorporate a number of GCs allocated at the decode time. Each GC contains a complete snapshot of all relevant processor state including the RAT. Recovery at an instruction using a GC is "instantaneous", i.e., it requires a fixed, low latency independent of the number of squashed instructions. Thus, the more instructions are squashed on each mispeculation, the more preferable the GC-based recovery will be over the ROB-based recovery (ignoring the effects of GCs on the operating frequency).

GC is a coarse-grain recovery mechanism as it allows recovery at some instructions. Ideally, for each instruction, a GC would be allocated. However, the larger the NoGCs, the longer the RAT delay, and thus the lower overall performance. Hence, using as few GCs as possible is desirable. Recovery at a specific branch without a GC can be done either by using ROB or by recovering at an earlier GC, and then re-executing all other instructions in between. GC prediction and selective GC allocation techniques can help achieve performance close to the performance achievable with an infinite GCs while implementing very few GCs in hardware [5] [3] [4].

4.1.5 Checkpointed RAT's Operations

The RAT supports the following operations: read (lookup), write (update), GC allocation and GC restoration. Section 4.1.1 discussed RAT reads and writes. The other two operations implement the RAT checkpointing function. With sRAT, a GC is taken by copying the whole RAT into one of the backups (GC allocation). With sRAT, RAT recovery is done by copying one of the RAT backups (GCs) to the main RAT (GC restoration). With cRAT, a GC is taken by copying the valid bits into one of the valid-set backups (GC allocation). With cRAT, RAT recovery is done by copying one of the valid-set backups (GCs) to valid bits (GC restoration).

4.1.6 sRAT: GCs and Performance

Earlier processors used few GCs (e.g., four) that were allocated at every predicted branch [70]. Few GCs were sufficient for the relatively small window size, containing few branches (e.g., window size of 32 instructions includes about six branches on average). However, modern processors use much larger windows, and hence have a lot more unresolved branches. Moreover, modern processors use other forms of speculation such as memory dependency prediction, and thus may require even more GCs. Previous work showed that 24 to 48 GCs would be needed to maintain high performance for processors with 128 or more instructions in their windows [46] . Assuming that embedding a large number of GCs into the RAT significantly increases RAT latency, previous work proposed using confidence estimators to selectively allocate GCs and throttle control flow speculation to achieve higher performance with four or fewer GCs [31] [5].

While previous studies have assumed that increasing the number of GCs degrades performance and energy, they have not quantified this performance degradation as a function of the number of

GCs. Instead, previous work relied on IPC measurements to estimate performance. However, IPC does not measure actual time. A technique that improves IPC may actually degrade performance if it reduces operating frequency. Accordingly, an open question is whether previous work has sufficiently reduced the number of required GCs and whether the conclusions are valid. This work complements previous RAT checkpointing work by quantifying the actual performance (execution time) taking into consideration both IPC and latency.

4.1.7 RAT Implementations: Related Work

Related work falls into two categories: The first category includes work focusing on measuring and modeling sRAT and cRAT energy and latency. Bishop et al., present an implementation for an sRAT with GCs for single-issue processors in a 350 nm technology and report its worst case delay [11]. De Gloria et al., report the latency of a 4-way superscalar sRAT with embedded cross-bundle dependence detection logic and a stack of four GCs in a 350 nm technology [21]. Buti et al. in [17] present the implementation details of the cRAT for POWER 4 processor. Our work complements previous work in that it studies RAT latency and energy as a function of several architectural parameters as opposed to focusing on a particular design.

The second category includes work focusing on reducing the number of sRAT GCs or RAT ports while maintaining performance (e.g., [34] [43] [57]). This work complements these studies by focusing on both IPC and latency. As the evaluation section shows, the tradeoffs are very different when actual sRAT latency is taken into consideration.

4.2 Physical-Level Implementation

This section presents two checkpointed sRAT designs representative of early and recent proposals respectively. These designs differ in the way GCs are organized and offer different levels of GC management flexibility. This section also discusses our checkpointed cRAT design.

4.2.1 Checkpointed RAT Designs

Figure 4-4 depicts the organization of a checkpointed RAT. Figure 4-4(a) shows the conceptual organization where multiple RAT copies exist. Figure 4-4(b) depicts how at the physical-level implementation, GCs can be interleaved and embedded into the RAT next to each main bit.

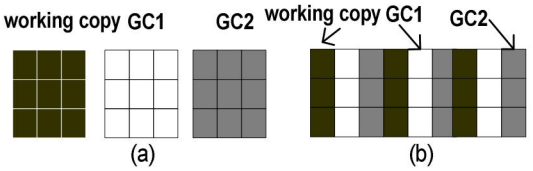


Figure 4-4: RAT Checkpointing: (a) Concept, (b) Implementation

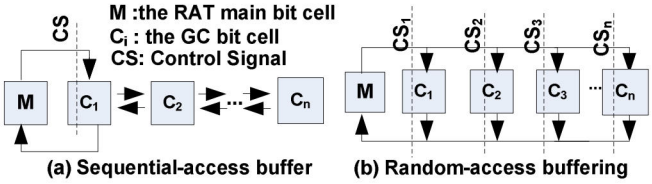


Figure 4-5: Checkpointing organizations: (a) SAB, (b) RAB

Although GCs provide low-latency recovery, they affect RAT latency and energy whether the GCs are embedded into or placed out of the RAT. This increase is primarily due to the load increase resulting from connecting GCs to the main RAT cell (discussed in Section 4.2.2).

Two checkpointed RAT designs have been assumed in previous work. These designs differ in the way they implement GC allocation and GC restoration. The first design organizes the GCs in a bi-directional shift register. The second design organizes the GCs in a random access buffer. For clarity, the terms SAB (Serial Access Buffer) and RAB (Random Access Buffer) will be used to refer to these implementations. SAB requires point-to-point connections, whereas RAB provides maximum GC management flexibility. Figure 4-5(a) and (b) show the organizations of the RAT cells for SAB and RAB implementations respectively. Every RAT main bit cell (marked as M) has 3xN read and N write ports; these ports are not shown in Figure 4-5. The GC cells are marked as C_i. In SAB, GC allocation is done by shifting the C_i bits to the right, copying the RAT bit value to the adjacent vacant position. In SAB, restoring from a GC may require multiple steps since the appropriate value must be shifted into the RAT main bit. For example, restoring from C_2 requires two left shifts. GC restoration in SAB may take multiple cycles depending on the number of GCs.

In RAB, the GCs are organized in a random access buffer; hence, GC allocation latency and GC restoration latency are nearly the same for all GCs. If the number of GCs becomes large, recovery using RAB is much faster than recovery using SAB since no shifting is needed. Both designs require an external controller to track the number of available GCs and to coordinate GC operations. For SAB, the controller keeps track of the number of GCs currently in the shift registers. For RAB, the controller tracks the status of each GC. SAB is more compact than RAB since it requires fewer external signals: RAB requires one read/write control signal per individual GC, while SAB only requires a global shift left/right signal.

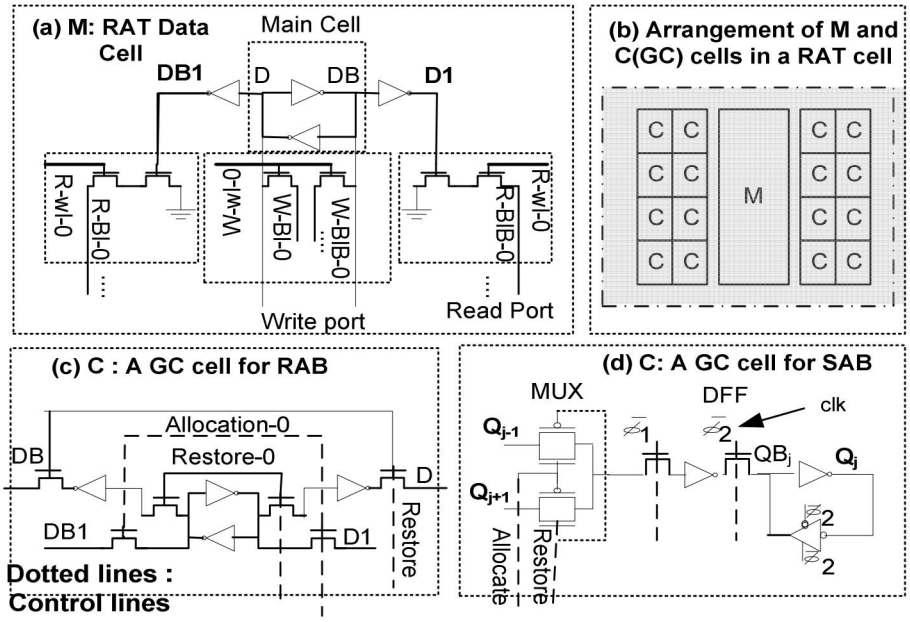


Figure 4-6: (a) RAT main cell (b) the layout of the main RAT bit and GCs, (c) RAB GC, (d) SAB GC cell

4.2.2 sRAT: Physical-Level Implementation

A non-checkpointed RAT is simply a multi-ported register file. A checkpointed RAT, however, is a multi-ported register file with embedded GCs. The checkpointed RAT circuit consists of the precharge and equalization circuitry, sense amplifiers, write drivers, control circuitry, decoders, along with an array of RAT cells connected by bitlines and wordlines. Figure 4-6(a) depicts the main RAT cell comprising two back-to-back inverters and several read and write ports. Figure 4-6(b) shows a complete RAT cell with 16 GCs. Each GC requires an SRAM cell. The SAB and RAB GC cells are shown in Figure 4-6(c) and (d) respectively.

The multi-ported sRAT cell uses one wordline and two bitlines per each write or read port. Multiple read operations may access the same RAT entry. In RAB, all GCs are connected via pass gate/gates to the main cell, whereas in SAB, only one GC is connected to the main cell directly. Hence, the main cell must be capable of driving a capacitance proportional to the number of ports and connected GCs. To protect the data stored in the main cell during multiple accesses, decoupling buffers isolate the RAT main cell and the read ports [26]. Since the GCs are connected to the main cell as the read ports do, the buffers also isolate the GCs. Due to these isolating buffers, separate write bitlines are required. Differential read and write operations are used because they offer better power, delay and robust noise margins. To reduce power, the following techniques are employed: *(i)* pulse operation for the wordlines, for the periphery

circuits and for the sense amplifiers; *(ii)* multi-stage static CMOS decoding; *(iii)* current-mode read and write operations.

Figure 4-6 (c) and (d) show RAB and SAB GC cells respectively. In SAB, GCs are organized as bi-directional shift registers with connections between adjacent cells; only one of the GCs is connected to the main RAT bit through pass gates. In SAB, a GC cell consists of a register and a multiplexer controlling the shift direction. The SAB's shift register uses two non-overlapping clocks. The SAB requires two external control signals irrespective of the number of GCs. In RAB, each GC cell is connected to the RAT main cell through separate pass transistors. Two pairs of pass transistors are used to copy the value from the RAT main cell to the GC and vise versa. Each RAB GC cell needs two external control signals.

4.2.3 cRAT: Physical-Level Implementation

This section discusses the implementation details of our cRAT design. A single-compare CAM cell can support read, write and compare for 1-bit data. Since single-cycle renaming requires many simultaneous lookups against each CAM entry in a clock cycle, a traditional cRAT implementation needs multi-compare cells. Figure 4-7 shows the organization of our cRAT implementation where rather than having an integrated multi-compare CAM cell, the compare and match are performed outside the CAM entry. The CAM content store and CAM content update operations are separated from the compare and match (or lookup). The CAM storage cells comprise the SRAM storage cells, write circuitries and match array drivers. The compare and match logic is physically separated from the CAM entry and placed in a match entry that is

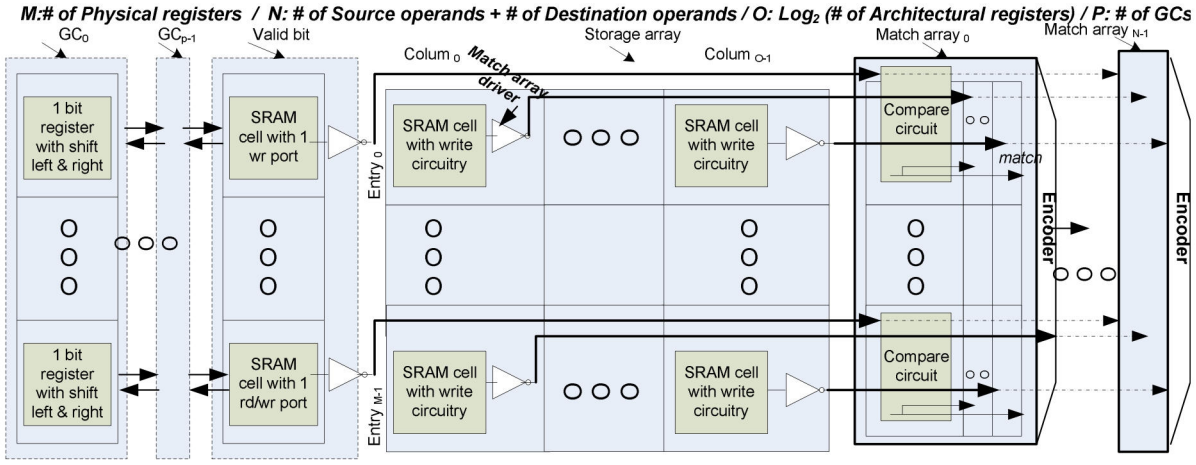


Figure 4-7: High-level organization of CAM-based RAT implementation

horizontally aligned with the CAM storage entry. The match arrays and CAM storage array have the same number of entries. The match array comprises compare and match logic for lookup operation. The number of simultaneous CAM lookups determines the number of match arrays. The bits in a CAM storage entry are driven horizontally to all match arrays. Each search pattern is transmitted to a separate match array, driven vertically on buses across the array entries. For a given match array, a match occurs when all bits of the search pattern match the corresponding bits of a CAM storage entry. Match array entries utilize the compare circuit depicted in Figure 4-8. Each match array uses an encoder to encode the matched physical register's corresponding address. The match lines of each match array drive the encoder and the readout circuits.

This cRAT design has several advantages over the one based on traditional multi-compare CAM cells. The latter incurs significant area, latency and energy overheads for running many compare circuits, match lines, search-lines and search-line-bars across each CAM entry. Moreover, traditional compare circuits present substantial wiring and device loads and demand larger storage cells capable of driving these loads. Moving compare circuits outside the CAM array significantly reduces the height/width of the CAM cell. In our design, the match line encoder is integrated locally within each match array. This technique reduces the total load on the match lines and the size of the match line drivers and bit-wise compares, and hence the overall width and area of the match array. This technique also increases match speed generation and encoding. The physical-level design of the storage part is similar to that of sRAT addressed in Section 4.2.

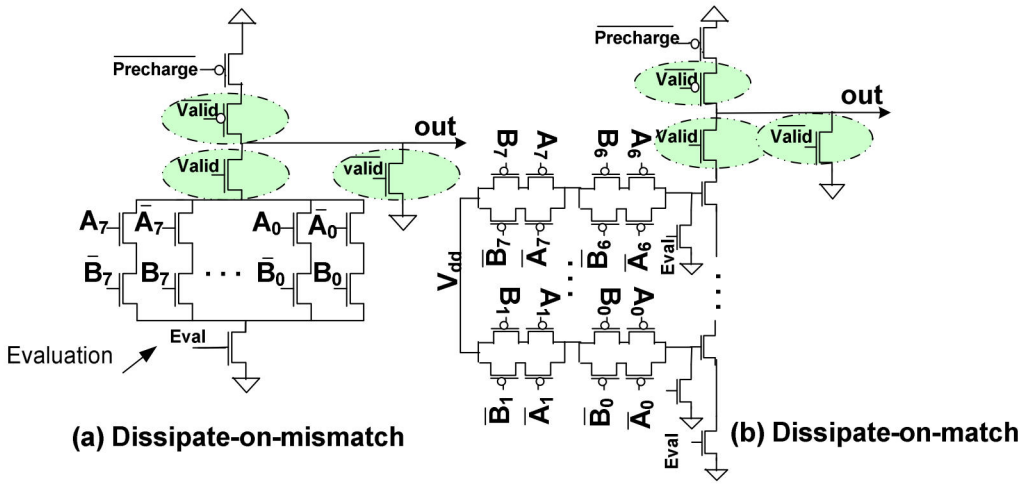


Figure 4-8: Comparators

In cRAT, only one comparator will match during a lookup. Traditional cRAT implementations used the dissipate-on-mismatch comparator depicted in Figure 4-8(a). With this comparator, all entries, but one, consume energy on accesses. The dissipate-on-match comparator, introduced in [24] and depicted in Figure 4-8 (b) (the circled transistors are not required), reduces energy at the cost of more area. We exploit the valid bits to further reduce energy. The motivation is that only entries with set valid bits should participate in lookups. The majority of entries will not have their valid bit set. This optimization requires three extra transistors per comparator. Figure 4-8 depicts the extra transistors (shown circled). If the valid bit is on, the normal compare operation will proceed; otherwise, neither precharge nor evaluation will be performed. In any case, the match line is set to zero through a pull-down NMOS transistor as it needs to deliver a mismatch output to the encoder. Our cRAT uses the comparator depicted in Figure 4-8(a) to avoid extra area penalty. Section 4.4.2.5 discusses the energy reductions achieved by this optimization. As Section 4.4.1 addressed, SAB is a more efficient technique since only one GC cell has a direct connection with the main valid cell. Furthermore, unlike RAB, SAB does not demand a large main cell. Hence, we used SAB GC organization for our checkpointed cRAT.

4.3 sRAT: Analytical Models

Since no analytical model exists for the checkpointed RATs, previous work relied on analytical models for register files. However, register file models do not consider the impact of GCs on latency and energy. This section presents analytical models for the worst-case latency and energy of the SRAM-based, checkpointed RAT implementation.

This section is organized as follows: Section 4.3.1 discusses the model-developing methodology and the model's input parameters. Section 4.3.2 presents the latency and energy models respectively. For energy models, the methodology of Section 3.3.3 is also applicable to RATs.

Table III : Analytical Model Input Parameters

Physical Level Organizational Parameters	
NoE	*Number of entries*
WoE	*Width of each entry*
NoRP	*Number of read ports*
NoWP	*Number of write ports*
NoGC	*Number of global checkpoints (GCs)*
Technology Parameters (Addressed in Table I)	

Section 4.4.2.4 of the evaluation section compares the model estimations against the circuit simulation results.

4.3.1 Methodology

Section 3.3.1 described our overall design methodology. For the base RAT, the transistors can be sized to achieve different speed/energy tradeoffs. In our implementation, we sized the transistors of the base 4-way and 8-way RATs (Section 4.4.1) such that the RAT read delay be less than the upper bound estimated by CACTI 4.2 [65]. In our models, the geometry and transistor sizes of the base RAT cells (cell port requirements are addressed in Section 4.1.2) are extracted from our full-custom layout.

The models do not account for external loads since these loads are independent of the RAT implementation. Although extending these models to predict the latency and energy for other technologies is feasible, it is beyond the scope of this work.

Table III lists the model's input parameters. The parameters fall under two broad classes: physical-level organizational parameters and technology-specific parameters. The physical-level organizational parameters are as follows: the number of entries, NoE, the width of each entry, WoE, the number of read ports, NoRP, the number of write ports, NoWP, and the number of GCs, NoGCs. In our analytical models, the relations among the physical-level organizational parameters and the architectural-level parameters are given by (35) to (38).

$$\text{WoE} = \log_2(\text{Number of physical registers}) \tag{35}$$

$$\text{NoE} = \text{Number of architectural registers} \tag{36}$$

$$\text{NoRP} = (\text{Number of source operands per instruction} + \text{Number of destination operands per instruction}) \times \text{IW} \tag{37}$$

$$\text{NoWP} = (\text{Number of destination operands per instruction}) \times \text{IW} \tag{38}$$

4.3.2 Delay Model

This section presents the analytical, worst-case delay model for the checkpointed, sRAT. For clarity, labels are assigned to the elements in the critical path. These labels are used as subscripts

to specify the corresponding resistance and capacitance. The type of gates (e.g., inverter) and the type of capacitors (e.g., drain: d, source: s, and gate: g) are also denoted in the subscripts.

Our RAT design is based on a synchronous multi-ported SRAM, i.e., a clock starts off the accesses. The SRAM consists of six main sub-blocks: a decoder to decode the input address; a memory core of bit cells arranged in rows and columns; a read logic comprising a read-column differential sense amplifier and output data drivers; a write logic to drive data onto the bitlines; and, read and write control logic to control the write and read logic respectively. The RAT read delay and RAT write delay are given by (39) and (40) respectively. The following subsections present the per component delay analyses.

$$T_{RAT\_read\_access} = T_{decoder} + T_{wordline} + T_{bitline} + T_{sense\_amplifier} \quad (39)$$

$$T_{RAT\_write\_access} = T_{decoder} + T_{wordline} + T_{bitline} + T_{write\_driver} \quad (40)$$

4.3.2.1 Component Delay: Decoder

A separate decoder is needed per read port and per write port. Figure 4-9 shows the decoder's high-level architecture, critical path and equivalent RC circuit. To estimate the RC delay, transistor sizes and interconnect lengths along the critical path are required. These parameters are a function of WoE, NoRP and NoWP. Increasing the number of read ports (NoRP), the number of write ports (NoWP) and the number of GCs (NoGCs) increases the RAT cell geometry and the interconnect lengths between subsequent rows and between subsequent columns.

The decoder has a hierarchical architecture. In the predecode stage, each 3-to-8 decoder generates a 1-of-8 code for every three address bits. If the number of address bits is not divisible by three, a 2-to-4 decoder or an inverter is used. Each x-to-2^x decoder consists of 2^x NAND gates and x inverters to complement the address inputs. In the second stage, the pre-decoder outputs feed NOR gates. The decoder delay is the interval between the moment the address input passes the INV (E1)' threshold voltage and the moment the NOR (E3)' output reaches the threshold voltage of the following NAND (E4). Equations (41) to (46) report the number of address bits (N_{addr}), the number of 3-to-8 decoders (N_{3to8}), the number of NOR gates (N_{nor}) and the fan-in of NOR gate ($N_{nor\text{-}input}$) as a function of the model's input parameters. Equations (43), N_{2to4}, and (44), N_{inv}, calculate if an additional 2-to-4 decoder or an inverter is required when N_{addr} is not

divisible by three. Equation (47), $N_{nor\text{-}a\text{-}nand}$, calculates the number of NOR gates driven by each NAND gate. Given by (48), the length of the wire between two NOR gates fed by a specific NAND gate of the predecode stage is a function of the RAT cell's height and NoGC. The corresponding resistance and capacitance are calculated by (49). Equations (51) to (53) calculate the time constants (τ) for the RC circuit depicted in Figure 4-9(c).

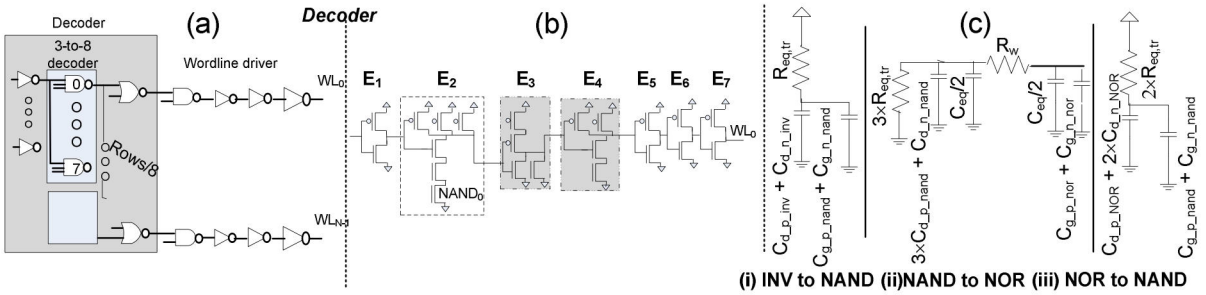


Figure 4-9: (a) Decoder and wordline driver, (b) Critical path, (c) Equivalent RC circuit

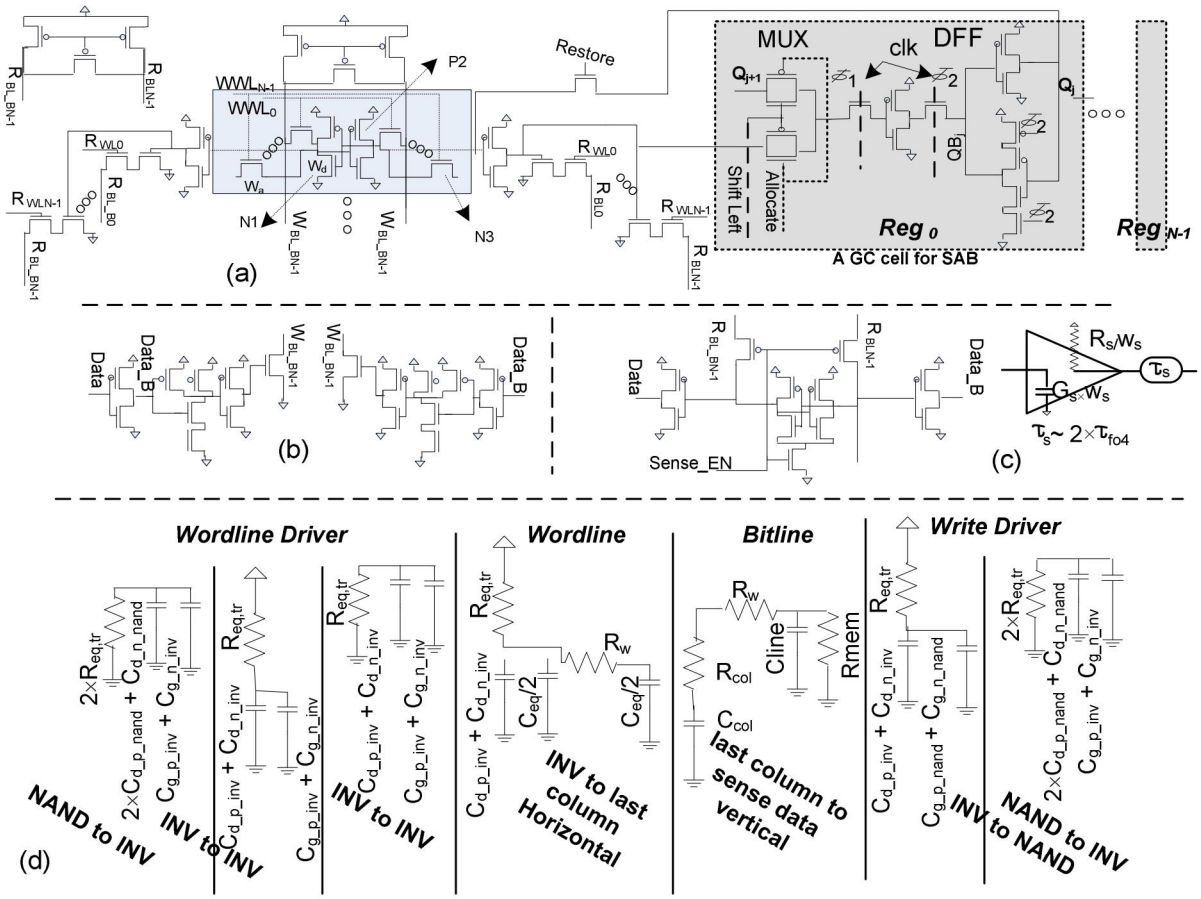


Figure 4-10: Principal building blocks (a) multi-ported SRAM cell and its connection with a SAB GC cell, (b) write driver, and (c) sense amplifier (d) RC equivalent circuits.

$N_{addr} = \lceil \log_2 (NoE) \rceil$ (41)

$N_{3to8} = \lfloor 1/3 \times (N_{addr}) \rfloor$ (42)

$N_{2to4} = \lfloor 1/2 \times [(N_{addr}) - 3 \times (N_{3to8})] \rfloor$ (43)

$N_{inv} = (N_{addr}) - 3 \times (N_{3to8}) - 2 (N_{2to4})$ (44)

$N_{nor} = NoE$ (45)

$N_{nor\text{-}inputs} = N_{3to8} + N_{2to4} + N_{inv}$ (46)

$N_{nor\text{-}a\text{-}nand} = NoE/8$ if (N_{addr} is divisible by 3) (47)

L_{cw} (μm) = wire length between two NOR gates fed by the same NAND gate of the pre-decoder.
$L_{cw} \approx (\text{Height of multi-ported cell}_{(4 \text{ way or } 8\text{way})} \times (1 + \Sigma_{i=0}^{i= NoGC} 0.01^i)) \times N_{nor\text{-}a\text{-}nand}$ (48)

$R_{cw}(Ohm) = R_{Ohm/\square} \times (L_{wire}/W_{wire})$, $C_{cw}(Farad) = C_{(Farad/um)} \times L_{wire}(m)$ (49)

$\tau_{INV\text{-}to\text{-}NAND} = R_p \times (C_{d\_p\_inv\_INV} + C_{d\_n\_INV} + C_{g\_p\_NAND} + C_{g\_n\_NAND})$ (50)

$C_{eq} = (C_{g\_p\_NOR} + C_{g\_n\_NOR}) \times N_{nand\text{-}to\text{-}nor} + L_{cw} \times C_{metal/um}$ (51)

$\tau_{NAND\text{-}to\text{-}NOR} = 3 \times R_n \times (C_{eq}/2 + 3 \times C_{d\_p\_NAND} + C_{d\_n\_NAND}) + (3 \times R_n + R_{cw}) \times C_{eq}/2$ (52)

$\tau_{NOR\text{-}to\text{-}NAND} = 2 \times R_p \times (C_{d\_p\_NOR} + 2 \times C_{d\_n\_NOR} + C_{g\_p\_NAND} + C_{g\_n\_NAND})$ (53)

4.3.2.2 Component Delay: Wordline Driver

Figure 4-9 shows the critical path along the wordline driver and the inverter chain following it. The equivalent RC circuit for the critical path is depicted in Figure 4-9(c). The NAND (E4) inputs are the decoder output and the *operation select* input (read, write, or "no operation"). The worst-case delay occurs when one of the NAND inputs turns off and the last column's pass transistor turns on. Each wordline driver consists of a NAND gate followed by an INV or an INV chain depending on the wordline capacitance. Wordline capacitance depends on the number of pass transistors along it, 2×NoE. Furthermore, the wordline capacitance depends on L_{cw}, the interconnect length between the wordline driver and the SRAM's last columns. L_{cw} is a function of NoGCs, the SAB cell's width, and the multi-ported SRAM cell's width. L_{cw} , given by (58), is used to estimate equivalent resistance, R_{eq}, and capacitance, C_{eq}, required in (59).

$\tau_{NAND\text{-}to\text{-}INV} = 2 \times R_n \times (2 \times C_{d\_p\_NAND} + C_{d\_n\_NAND} + C_{g\_p\_INV} + C_{g\_n\_INV})$ (54)

$\tau_{INV\text{-}to\text{-}INV} = R_p \times (C_{d\_p\_INV} + C_{d\_n\_INV} + C_{g\_p\_INV} + C_{g\_n\_INV})$ (55)

$\tau_{INV\text{-}to\text{-}INV} = R_n \times (C_{d\_p\_INV} + C_{d\_n\_INV} + C_{g\_p\_INV} + C_{g\_n\_INV})$ (56)

$C_{eq} = (2 \times C_{gate\_pass}) \times WoE + L_{cw} \times C_{metal/um}$ (57)

$$L_{cw} = (\text{width of multi-ported cell}_{(4\text{-way or } 8\text{-way})} + \text{width of a SAB GC cell} \times NoGCs) \times WoE \quad (58)$$

$$\tau_{INV\text{-}to\text{-}Last\_Columns} = R_p \times (C_{eq}/2 + C_{d\_p\_INV} + C_{d\_n\_INV}) + (R_{eq,tr} + R_{wire}) \times C_{eq}/2 \quad (59)$$

4.3.2.3 Component Delay: Bitline Delay

This subsection discusses the components contributing to the bitline delay for read and write operations. The reading from or writing to a row is preceded by precharging all bitlines (BLs and BLBs) to $V_{precharge}$, and the selection of a row by the decoder. The bitline precharge time is designed to be hidden under the address decoding time to achieve shorter read/write access time. As shown in Figure 4-10, a wordline and a set of BLs/BLBs are selected to drive the contents of the memory cell(s) to the sense amplifier for a read operation, or to the write driver(s) for a write operation. During a read, when the wordline goes high, one of the pull-down transistors of the back-to-back inverters will begin to conduct, discharging BL or BLB. Column isolation PMOS transistors are turned on to allow the voltage difference between BL and BLB to develop to the sensing voltage (V_{sense}), helping the amplifiers to quickly sense the data. The BL delay of the read operation is the time interval between the moment the wordline goes high and the moment one of the bitlines reaches the voltage V_{sense} below its maximum value $V_{precharge}$.

The delay of the latch-based sense amplifier, depicted in Figure 4-10(c), consists of the latch delay and the buffer delay. The latch delay depends on the voltage gain and the BL swing's speed. The latch's amplification delay is proportional to the logarithm of the required gain and the load on the amplifier outputs [9]. For a gain of about 20 with only the self loading of the sense amplifier, [9] reports an amplification delay of about two fanout-of-4 (FO4) inverter delays. We use the same estimation for the sense amplifier delay in our models.

Write drivers pull down the pre-charged BL/BLB to zero. As depicted in Figure 4-10(b), two NMOS transistors connect the write circuits to the write BL/BLB during write cycles. The write driver's critical path consists of an INV, a NAND and another INV. The final inverter feeds the gate of an NMOS isolator transistor. The worst-case BL delay for a write operation is the time interval between the moment that wordline goes high and the moment the cell content is inverted (V_{bit} should be a very low voltage near zero). Figure 4-10(d) depicts the RC equivalent circuit along the critical path for read and write operations ((60) to (69)).

$$R_{mem} = R_{n,\,Wd} + R_{n,\,access} \quad (60)$$

$$C_{bitline} = 2 \times C_{d\_p\,(Wbitpre)} + NoE \times (1/2 \times C_{d\_n\_pass(Wa)}) + (\text{Height of multiported cell}_{(4\text{ way or }8\text{way})} \times (1 + \Sigma_{i=0}^{\,i=NoGC}\, 0.01^{i}) \times C_{\,metal/um}) \quad (61)$$

$$R_{col\_read} = R_{eq,\,p}\,,\ R_{col\_write} = R_{eq,\,n} \quad (62)$$

$$C_{col\_read} = C_{d\_p\_access\_SA} \quad (63)$$

$$C_{col\_write} = C_{d\_n\_write\_driver} \quad (64)$$

$$\tau_{Last\_Columns\_to\_Sense\ amplifier}\ \textbf{(or}\ \tau_{Last\_Columns\_discharge(for\ write)}\textbf{)} = [R_{mem} \times C_{bitline\_write} + (R_{mem} + R_{bitline\_write} + R_{col}) \times C_{col})] \times \ln(V_{precharge}/(V_{precharge} - V_{bit})) \quad (65)$$

(For more detail refer to the RC tree analysis [69].)

$$\tau_{Senseamplifier} = 2 \times \tau_{fo4} \quad (66)$$

$$\tau_{\,INV\text{-}to\text{-}NAND} = R_{\,eq,\,tr}\ (C_{d\_p\_inv\_INV} + C_{d\_n\_INV} + C_{g\_p\_NAND} + C_{g\_n\_NAND}) \quad (67)$$

$$\tau_{NAND\text{-}to\text{-}INV} = 2 \times R_{\,eq,\,tr}\ \times (\,2 \times C_{d\_p\_NAND} + C_{d\_n\_NAND} + C_{g\_p\_INV} + C_{g\_n\_INV}) \quad (68)$$

$$\tau_{INV\text{-}to\text{-}nmos} = R_{\,eq,\,tr} \times (\,C_{d\_p\_INV} + C_{d\_n\_INV} + C_{g\_n\_INV}) \quad (69)$$

4.3.2.4 Operation Delay

The switching delay from input to output is referred to as propagation delay that is the time required for the output to reach 50% of its final value, V_{dd}, when the input changes. The output follows an exponential trend ($V_{out}(t)=V_{dd} \times e^{-(T/\tau)}$); and, the time it takes for the output to reach $V_{dd}/2$ is $\ln(2) \times \tau$. The read and write delays are given by (70) and (71) respectively, where the time constants, $\tau_{\#}$, correspond to the equations with the same numerical subscript.

$$Delay_{\,Read} = \ln(2) \times (\tau_{17} + \tau_{18} + \tau_{19} + \tau_{20} + \tau_{21} + \tau_{22} + \tau_{25} + \tau_{31} + \tau_{32}) \quad (70)$$

$$Delay_{\,Write} = \ln(2) \times (\tau_{17} + \tau_{18} + \tau_{19} + \tau_{20} + \tau_{21} + \tau_{22} + \tau_{25} + \tau_{31}) \quad (71)$$

4.4 Evaluation

This section discusses the results of our physical-level and architectural-level evaluations.

4.4.1 sRAT: Physical-Level Evaluation

4.4.1.1 Design Assumptions and Methodology

The base 4-way RAT has 12 read ports and 4 write ports; and, the base 8-way RAT has 24 read

ports and 8 write ports. These base RAT configurations do not include GCs. We also assume that 64 architectural registers are available, typical of modern load/store architectures. Modern processors have about 128 physical registers. However, future processors may include more physical registers to support larger scheduling windows and/or multiple threads. Hence, the number of physical registers is varied from 128 to 512. We focus on RAT designs with 0, 4, 8 or 16 GCs since previous work shows that with GC prediction and selective GC allocation, 16 GCs are sufficient to achieve performance close to the performance achievable with an infinite number of GCs [3]. We developed full-custom layouts for both sRAT and cRAT using the Cadence® tool set in a commercial 130 nm fabrication technology with a 1.3V supply voltage. For circuit simulations, we used Spectre™, a vendor-recommended simulator. In the result section, worst-case delay and energy values are reported.

Circuit designs can be tailored to achieve different latency and energy tradeoffs. In an actual commercial design, a target latency and/or energy is decided and used as a specification for tuning the individual components. In lieu of an actual specification for the target operating frequency, we used CACTI 4.2 to obtain a reasonable upper bound on delay. CACTI is an integrated cache access time, cycle time, area and power modeling tool that is commonly utilized by computer architects. Using CACTI, for the base 4-way RAT, we determined an upper bound on the critical path delay by estimating the delay of a 64-bit, 64-entry SRAM with 12 read ports and 4 write ports. Similarly, we estimated the delay of a 64-bit, 64-entry SRAM with 24 read ports and 8 write ports to determine an upper bound on the critical path delay for the base 8-way RAT. These upper bounds are reasonable approximations since the base non-checkpointed RAT designs are identical to register files. However, the data width of the register files modeled by CACTI is larger. To further corroborate these RAT delay estimations, we also considered the clock periods of processors built in 130 nm fabrication technology (e.g., 800 MHz SR71010B MIPS) assuming that a single-cycle register renaming has been used.

4.4.1.2 sRAT: Latency

Figure 4-11 reports sRAT read and write latencies for RAB and SAB as a function of IW, WS and NoGCs. Figure 4-11(a) and (b) show the latencies of the 4-way and 8-way superscalar sRATs respectively. As expected, RAT delay increases with increasing WS, IW and NoGCs for both implementations. Adding more GCs increases delay, more so for RAB. The read and write

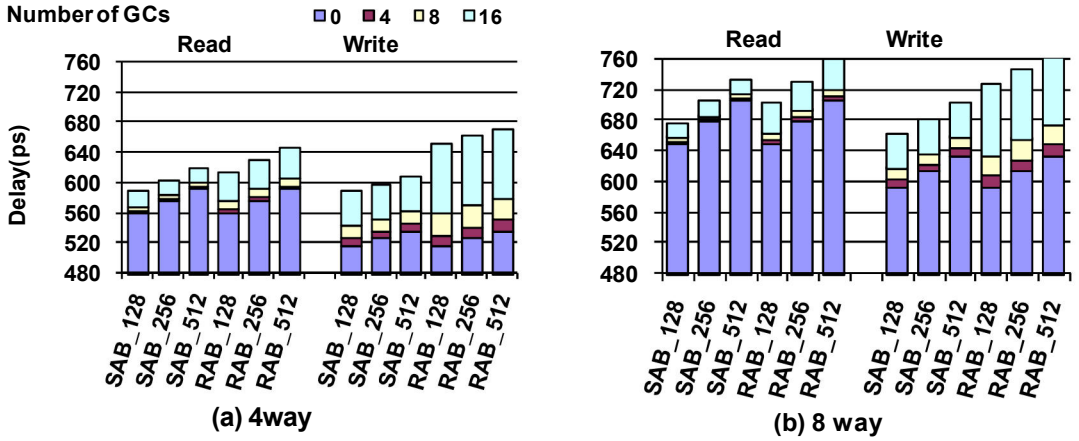


Figure 4-11: RAT read delay and RAT write delay as a function of the NoGCs with window sizes of 128, 256, and 512, (a) 4-way (b) 8-way

operations of RAB are slower than those of SAB because in RAB the main RAT bit is connected via pass transistors to all GC bits. The increase due to additional GCs is more pronounced for the 4-way superscalar RAT. The delay of the 8-way RAT, however, is dominated by the load of the extra read ports, and hence GCs have less impact on overall delay.

To comment on sRAT delay variation as a function of NoGCs, we focus on SAB- and RAB-512. We compare the read and write delays for sRATs with four, eight or 16 GCs with those of the RAT with no GCs. SAB RAT reads are 0.5%, 1.6% and 5% slower depending on NoGCs; SAB RAT writes are 1.8%, 4.8% and 13.8% slower. RAB RAT reads are 0.8%, 2.7% and 9.3% slower, whereas RAB RAT writes are 2.7%, 8% and 25.3% slower respectively. We observed that a SAB RAT with 16 GCs is faster than a RAB RAT with 4 GCs. These results suggest that a RAB RAT must improve IPC considerably over a SAB RAT to improve real performance.

4.4.1.3 Operating Frequency

In the simplest possible implementation, the RAT operates at the same frequency as the processor. In this implementation, instructions read from the RAT and then write new mappings back to it within a single cycle. Alternatively, the RAT can be pipelined to achieve a higher operating frequency at the expense of increased complexity and hardware cost. This work focuses on single-cycle register renaming as this represents a reasonable and common design point. To further support the validity of this assumption, we present sRAT latency in terms of the FO4 inverter delay. For instance, consider the 4-way sRAT with 12 read and four write ports and no GCs. For the 130 nm fabrication technology that we used, the FO4 delay, measured by

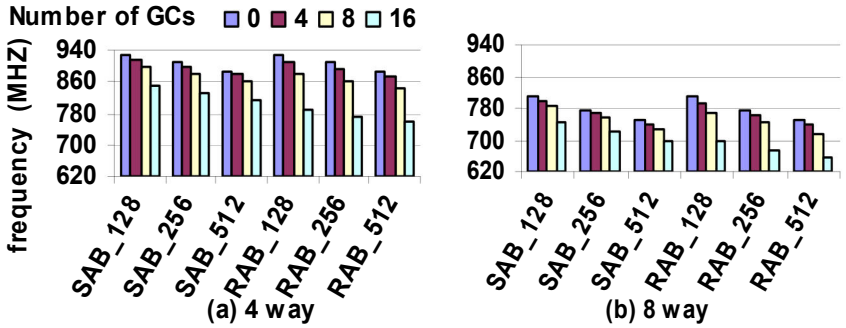


Figure 4-12: Clock frequency as a function of the number of GCs for window sizes: 128, 256, 512 (a) 4-way (b) 8-way.

simulations, is about 40ps. For the SAB sRAT, 12 simultaneous reads followed by four simultaneous writes take 1077ps (27 FO4) assuming a non-pipelined sRAT. Overlapping the decoding of the RAT write with that of the preceding RAT read can reduce the delay to 20 FO4. This delay is comparable to the clock period of the processors used as a specification for our design (e.g., 800MHz SR71010B MIPS).

Figure 4-12 shows the maximum operating frequency for the 4-way and 8-way RATs assuming that the RAT latency is the same as the clock period and the latching overheads are ignored. As observed, for a given WS and IW, RAB's performance deteriorates more rapidly than SAB's performance as the number of GCs increases.

4.4.2 sRAT: Architectural-Level Evaluation

This section discusses the effect of the architectural-level GC management policies on performance and energy for the checkpointed, SRAM-based RAT implementation.

4.4.2.1 Methodology

We used Simplescalar v3.0 [16] to simulate the processors detailed in Table IV. We study 4- and 8-way dynamically-scheduled superscalar processors with 128-, 256- or 512-entry window sizes. We compiled the SPEC CPU 2000 benchmarks for the Alpha 21264 architecture by HP's compilers and for the Digital Unix V4.0F using the SPEC-suggested default flags for peak optimization. All benchmarks were run using a reference input data set. The following SPEC CPU 2000 benchmarks are used in the experiments: ammp, applu, apsi, art, bzip2, crafty, eon, equake, facerec, fma3d, galgel, gap, gcc, gzip, lucas, mcf, mesa, mgrid, parser, swim, twolf, vortex, vpr and wupwise. We could not run the rest of the SPEC benchmarks either because we

TABLE IV : BASE PROCESSOR CONFIGURATION	
Branch Predictor	***Fetch Unit***
8K-entry GShare and 8K-entry bi-modal 16K selector, 2 branches per cycle	**Up to 4 or 8 instr. per cycle, 64-entry Fetch Buffer, Non-blocking I-Cache**
Issue/Decode/Commit	***Scheduler***
any 4 or 8 instr./cycle	**128, 256 or 512-entry/half size LSQ**
FU Latencies	***Main Memory***
Default simplescalar values	**Infinite, 200 cycles**
L1D/L1I Geometry	***UL2 Geometry***
64KBytes, 4-way set-associative with 64-byte blocks	**1Mbyte, 8-way set-associative with 64-byte blocks**
L1D/L1I/L2 Latencies	***Cache Replacement***
3/3/16 cycles	**LRU**
Fetch/Decode/Commit Latencies	
4 cycles + cache latency for fetch	

were not able to compile them or because they exhausted the available memory space during simulation. To achieve reasonable simulation times, samples were taken for one billion committed instructions per benchmark. Prior to collecting measurements, two billion committed instructions were skipped. Unless otherwise noted, we report the average over all benchmarks.

We limit our attention to two representative GC management policies. The first one, SEL, selectively allocates GCs only to low-confidence branches [1] [33]. Low-confidence branches are identified using a confidence estimator comprising a 1K-entry table of 4-bit resetting counters [31]. The second policy, ALL, allocates GCs to all branches. Both use in-order GC allocation and de-allocation. Unless otherwise stated, all performance results are normalized over a RAT with infinite GCs that are allocated at all branches and can be accessed in a single-cycle. Ignoring secondary effects, this RAT represents an upper bound on performance for the two GC allocation policies. We pessimistically assume that SAB can only shift by one bit per cycle. This assumption is valid for a simple SAB design. However, multiple shifts per cycle may be possible, and hence execution time could be better at the cost of increased complexity.

4.4.2.2 IPC Performance and Execution Time

Figure 4-13(a) shows the average IPC deterioration for the 4-way processor assuming that all RAT implementations operate at the same frequency. These measurements ignore the actual

implementation delay and compare just the IPC. For clarity, Figure 4-13 excludes the results for the non-checkpointed RAT. Performance deterioration with a non-checkpointed RAT is on the average 5%, 9.4% and 15% for the 128-entry, 256-entry and 512-entry windows respectively.

Irrespective of the GC allocation policy (SEL or ALL), with RAB, performance improves as the NoGCs increases since the recovery latency of all GCs is nearly the same. As previous work reported, performance is better with SEL than with ALL when few GCs are used since SEL allocates GCs judiciously to branches that more likely trigger recoveries. With ALL, however, GCs are allocated to all branches indiscriminately, and hence GCs get exhausted more often.

The same observation applies to SAB, where SEL still performs better than ALL with four or eight GCs. However, with SAB, performance does not always increase with increasing NoGCs. Specifically, performance degrades when NoGCs increases to 16 from eight because the number of cycles required to recover from a specific GC varies depending on the location of the GC in the SAB's shift register. As NoGCs increases, so does the expected number of shifts to retrieve that specific GC, and hence the cycles needed to restore from it. SEL requires more cycles for recovery as NoGCs is increased to 16 because when SEL cannot find a GC associated with a specific mispredicted branch, it restores at a later GC, and then walks back using ROB. The results show that RAB outperforms SAB if implementation latencies are ignored.

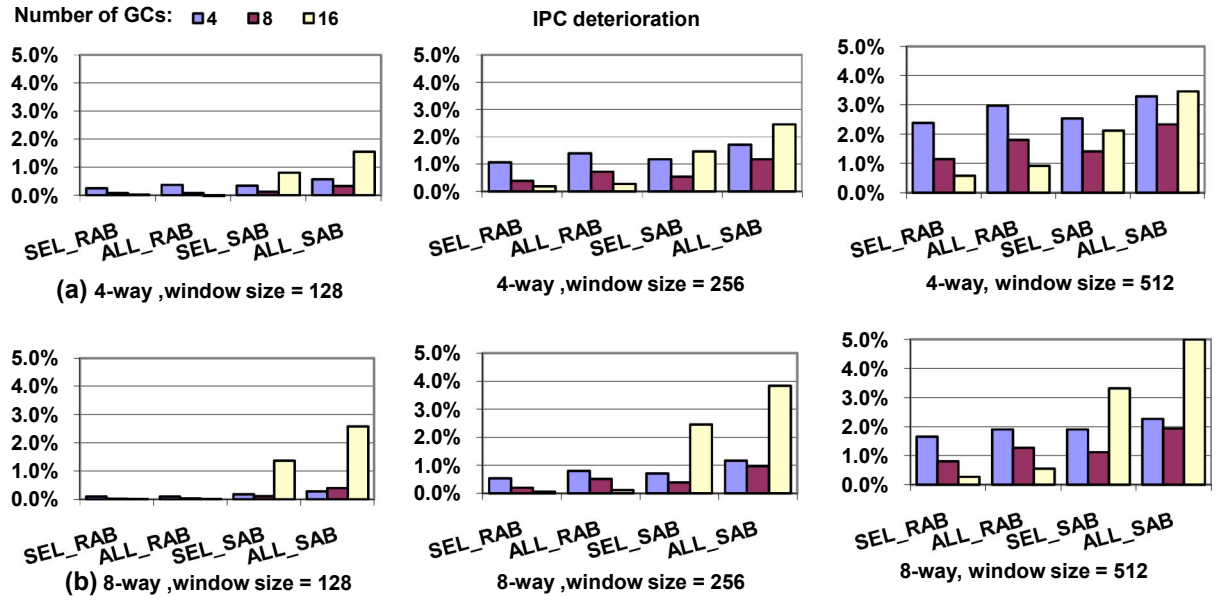


Figure 4-13: IPC deterioration compared to a design that uses an infinite number of GCs (a) 4-way and (b) 8-way (We assume that all designs operate at the same frequency)

Figure 4-13 (b) shows IPC deterioration for the 8-way superscalar processor. Wider (8-way) processors can fill the window faster than 4-way processors. Hence, they are more likely to speculate further down the instruction path. However, the deeper the speculation, the less likely it will succeed. Hence, additional GCs rarely will be useful. In most cases, they only serve to introduce additional delay while shifting the right GC to the main RAT cell in the SAB RAT.

Figure 4-14 reports execution time in seconds taking implementation delay into consideration. For this experiment, we assume that the sRAT delay determines the processor's clock period. To comment on the execution-time variation as a function of the NoGCs, we focus on the 4-way RAT for a 512-entry WS, shown in Figure 4-14 (a). Irrespective of the GC allocation policy, going from four to eight GCs, we observe up to 2.24% and 0.94% increase in total execution time for RAB and SAB respectively. Further, going from eight to 16 GCs, we observe up to 10.42% and 6.92% increase in total execution time. We observe similar variation trends for other window sizes. Thus, when we consider the delay overhead introduced by additional GCs, IPC does not correctly predict actual performance. In particular, IPC measurements suggest that performance always improves by increasing the number of GCs. Our experiments show that as predicted by IPC measurements, four or eight GCs improve overall performance compared to the case where no GCs exist, and recovery must exclusively be done using ROB. However, contrary to IPC prediction, the increase in the RAT latency with 16 GCs outweighs the IPC benefits. Furthermore, while eight GCs typically offer better IPC performance than four GCs, absolute performance in most cases deteriorates.

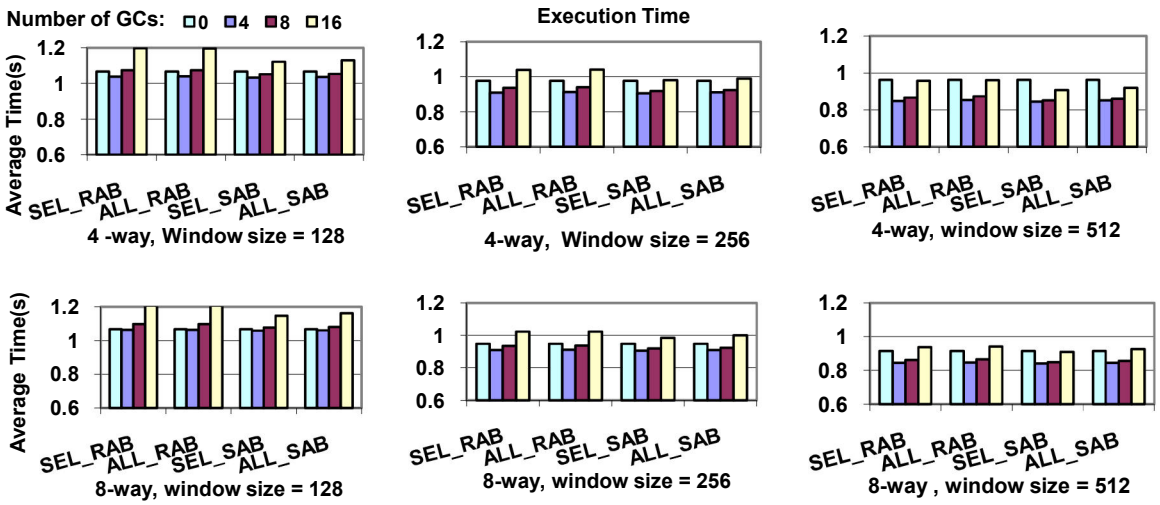


Figure 4-14: Execution time for: (a) 4-way, (b) 8-way superscalar processors.

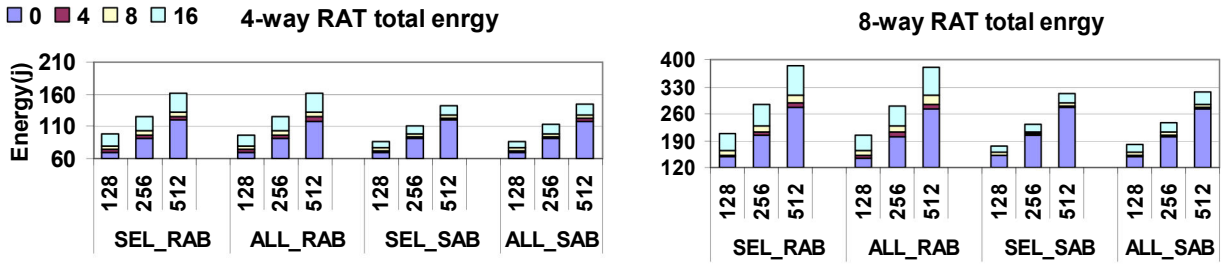


Figure 4-15: Total energy of RAB and SAB for both SEL and ALL methods (a) 4-way (b)8- way sRAT

As expected, the best design focusing on IPC alone would be different from the best design focusing on both IPC and latency. IPC measurements predict that the best performance is achieved by SEL_RAB with 16 GCs, whereas the best execution time is achieved by SEL_SAB with 4 GCs when the implementations' worst-case delay is taken into consideration.

The results show that ignoring the actual latency of the RAT incorrectly predicts performance, and performance does not monotonically increase with increasing NoGCs as IPC measurements suggest. Two components contribute to determining performance, and these components are at odds with each other. First, as more GCs are introduced, fewer cycles are spent recovering from mispeculations, hence improving performance. Second, introducing more GCs increases RAT latency, and consequently increases the clock period and decreases performance. In most cases, using very few GCs (e.g., four) leads to optimal performance.

4.4.2.3 sRAT: Energy

Figure 4-15 shows the average total sRAT energy as a function of NoGCs and WS for both SAB and RAB as well as GC management policies. Each energy value totals the energy for the RAT reads, RAT writes, GC allocations and GC restorations. SAB consumes less energy than RAB since more than 90% of RAT accesses are reads and writes, for them SAB consumes less energy than RAB. GC allocation and restoration of SAB are more energy consuming than those of RAB; however, these operations are relatively infrequent (10% of the total RAT accesses).

4.4.2.4 sRAT: On the Accuracy of the Analytical Models

Figure 4-16 (a) and (b) compare the circuit measurements with the analytical model estimations for energy and latency as a function of NoGCs. The worst-case relative error per operations is also depicted. The worst-case relative errors for energy and latency are, respectively, within

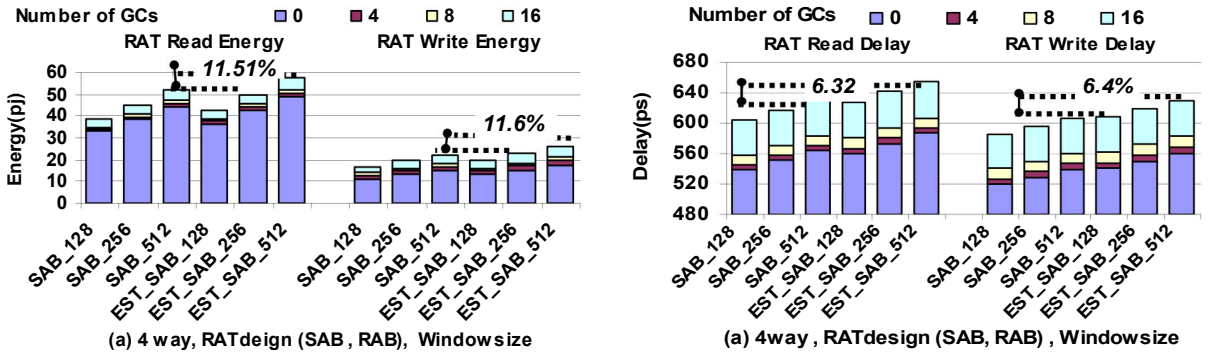


Figure 4-16: (a) Energy, and (b) Delay as a function of NoGCs and window sizes for the 4-way RAT (Simulation results and model estimations).

11.6% and 6.4% of the Spectre™ simulation results. The errors are monotonic, and the model estimations are in agreement with the physical-level simulation results in predicting latency and energy variation trends. The analytical model estimations differ from the circuit simulation results for the following reasons. Comparisons of the model-estimated and the layout-extracted capacitances show that about 5.1% of the error for energy is due to capacitance estimation inaccuracy. The formulas used to calculate gate and diffusion capacitances are over-simplified, and the capacitances are assumed to be voltage independent [41]. Each stage in our models (e.g., bitline, wordline) assumes that the inputs to the stage are step waveforms; however, actual waveforms in memories are far from steps, hence impacting the delay of a stage. The energy model exhibits a worst-case error of about 11.6%. Inaccurate capacitance estimation plays a key role in dynamic power consumption. The leakage power model also accounts for 5.7% of this error. Leakage current largely depends on the state of the circuit and temperature. Hence, it is difficult to quantify the leakage power accurately without circuit simulations.

4.4.2.5 sRAT: Simplified Latency Models

We also developed empirical delay and energy models for the checkpointed sRAT. To develop the empirical models, we applied curved fitting over 60 data points, based on the measurements from our physical-level implementations in 130 nm technology [8]. We present the latency models for the 4-way sRAT; equations (72) and (73) estimate SAB read and write latencies, while equations (74) and (75) estimate RAB read and write latencies. The worst-case relative error of the model predictions is on average within 7.8% of the Spectre™ circuit simulation results for the data points we used for curve fitting (0<=WS<=512, 0<=NoGCs<=16, IW=4 or 8). These data points cover values of the WS, IW and NoGCs that would be typical for modern

processors, and model predictions have a rage of accuracy that is acceptable for architectural-level studies. For predicting the delay and energy of the configurations outside of this range, extrapolation can be used. However, the extrapolation results are often subject to greater uncertainty. The formulas for quick estimation of the 4-way RAT delay as a function of NoGCs and WS are as follows:

$$\text{Delay (Read/SAB)} = 1.7834 \times e^{0.1632 \times \text{NoGCs}} + 23.07 \times \ln(\text{WS}) + 449.31 \quad (72)$$

$$\text{Delay (Write/SAB)} = 5.6727 \times e^{0.1432 \times \text{NoGCs}} + 14.236 \times \ln(\text{WS}) + 449.31 \quad (73)$$

$$\text{Delay (Read/RAB)} = 2.7831 \times e^{0.1763 \times \text{NoGCs}} + 23.05 \times \ln(\text{WS}) + 449.31 \quad (74)$$

$$\text{Delay (Write/RAB)} = 7.9821 \times e^{0.1654 \times \text{NoGCs}} + 14.215 \times \ln(\text{WS}) + 449.31 \quad (75)$$

4.4.3 cRAT: Physical-Level Evaluation

4.4.3.1 cRAT Latency and Energy

Figure 4-17 (a) reports lookup and update latency and energy for cRAT as a function of IW, WS and NoGCs. As WS increases, so does the number of entries, cRAT size, and hence lookup and update latencies. The observed trend is logarithmic. Increasing IW increases the number of write ports for the CAM storage cell, and hence its geometry. Moreover, it increases the number of match arrays and demands larger CAM storage cells capable of driving larger loads. Increasing IW and WS increase lookup and update latencies significantly. However, for a fixed IW and WS, increasing NoGCs does not have a noticeable impact on lookup and update latencies. Finally, energy increases significantly with WS. The observed trend is exponential.

Figure 4-17 (b) report lookup and update latency and energy for sRAT as a function of IW, WS and NoGCs. Increasing WS increases RAT delay logarithmically. However, energy increases only slightly. Adding more GCs increases sRAT latency because the bitline and wordline delays increase. For a fixed WS and IW, the latency and energy increase exponentially with increasing NoGCs. Increasing IW increases RAT read and write ports, and hence significantly increases latency and energy. The observed trend follows a quadratic polynomial line.

4.4.3.2 cRAT Lookup's Energy Optimization

Applying the proposed energy optimization to a 6-bit, 128-entry cRAT reduces the lookup energy by 40%. For 6-bit, 512-entry cRAT, we observe up to 58% lookup energy reduction. This reduction is not a linear function of the number of physical registers. The more the physical registers, the more comparators and extra switches are needed. Although these switches help decrease the dynamic energy, they increase area penalty and leakage power.

4.4.4 Comparing cRAT and sRAT

sRAT tends to be faster and more energy efficient for a small number of GCs. The difference between sRAT and cRAT grows larger with increasing the number of physical registers (or WS) for a fixed NoGCs. For example, for a 4-way, non-checkpointed sRAT, lookup requires 33% less energy and is 12% faster than cRAT lookups when the number of physical registers is 128. The differences grow to 416% for energy and 35% for latency with 512 physical registers.

The speed and energy difference between sRAT and cRAT grows smaller with increasing NoGCs for a fixed number of physical registers (or WS). Moreover, the less the physical registers, the less the difference. For example, 4-way sRAT lookups require 33.3% less energy and are 12.3% faster than cRAT lookups when the number of physical registers is 128, and the NoGCs is four. The difference shrinks to 23% for energy and -9% for latency when NoGCs increases to 32. When NoGCs passes a limit, cRAT becomes faster than its equivalent sRAT. For a constant number of architectural registers, the more the physical registers, the higher the limit. For instance, with 64 architectural and 128 physical registers, having more than x GCs (x >= 20) makes a 6-bit, 128-entry cRAT with x GCs faster than its equivalent 7-bit, 64-entry sRAT.

We observed that more than 90% of RAT accesses are lookups and updates. The latency and energy for GC allocation and restoration of sRAT are higher than those of cRAT. However, in most cases sRAT makes common case, lookups and updates, faster and more energy efficient.

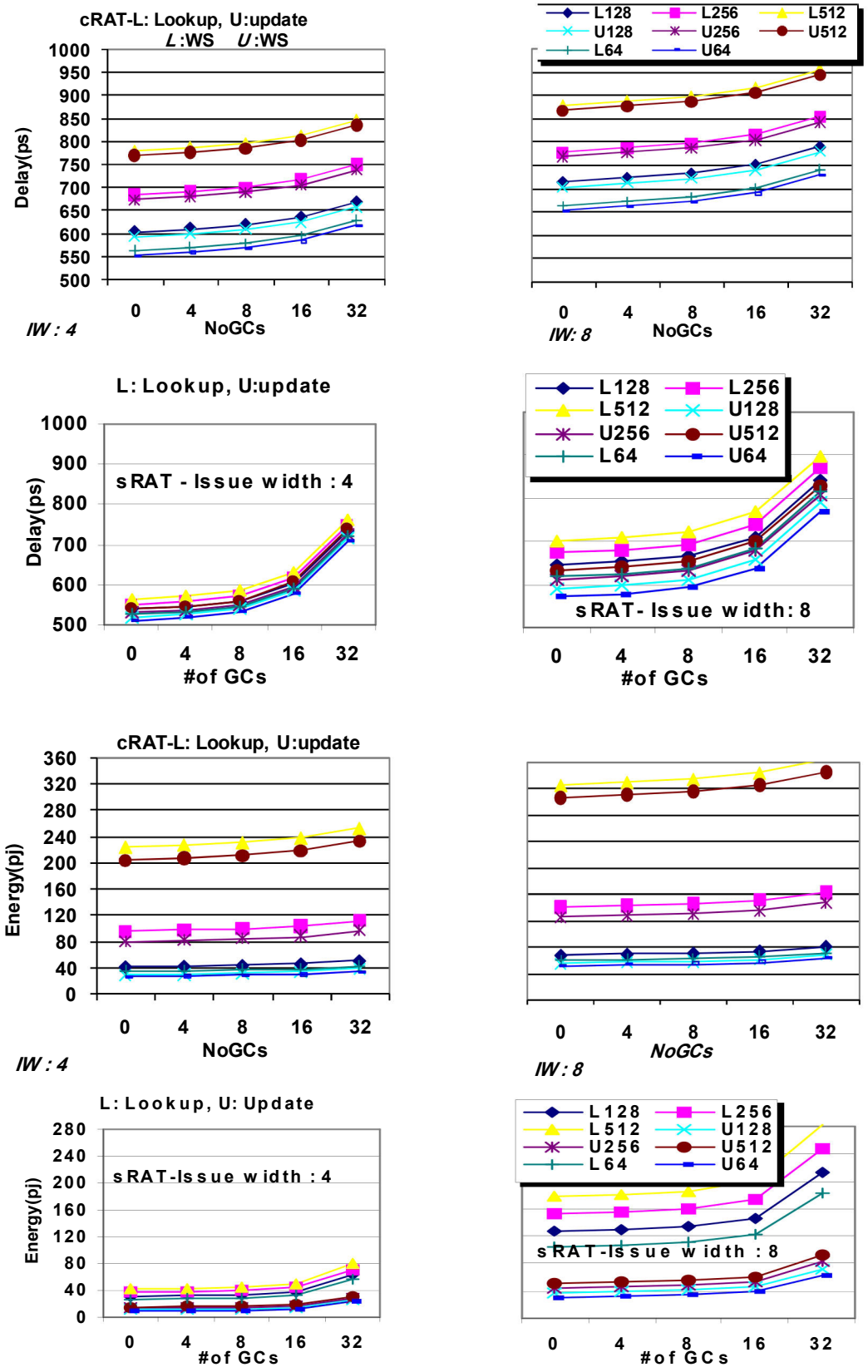


Figure 4-17: Lookup and update delay and energy as a function of the number of GCs for 128, 256, and 512 window sizes and 4-way and 8-way RATs (a) cRAT, (b) sRAT

4.5 Conclusion

Previous RAT checkpointing work developed GC count reduction techniques focusing solely on IPC performance evaluation to compare alternatives. Although previous work assumed that increasing the number of GCs increases the RAT delay, in performance evaluation, they ignored the effect of increasing the number of GCs on RAT delay and clock period, and hence did not accurately measure performance. This work improves upon previous work by determining quantitatively how RAT delay and energy vary as a function of IW, WS and GC count, utilizing full-custom checkpointed RAT implementations in a 130 nm fabrication technology. IPC performance evaluations show that performance improves monotonically by introducing more GCs. However, this work demonstrates that when RAT delay is taken into consideration, actual performance (execution time) rarely improves with more than four GCs for the SRAM-based, checkpointed RAT. This work has also studied two different GC management mechanisms, RAB and SAB. Although RAB, representative of recent checkpointed RAT designs, offers better IPC performance, SAB offers better actual performance (execution time) since SAB offers faster RAT reads and writes. Additionally, this work presents empirical and analytical latency and energy models for checkpointed SRAM-based RATs. Analytical models help computer architects estimate the latency and energy of various checkpointed RAT organizations during architectural-level exploration, where physical-level implementation is unavailable or unaffordable to develop. Comparisons show that the estimations provided by the models are in satisfying agreement with the simulation results. This work has also studied and compared the variation trends of the latency and energy for the checkpointed sRAT and cRAT as a function of the issue width, the window size and the number of GCs. Compared to sRAT, cRAT does not scale well with increasing the number of physical registers (or equivalently window size); however, it is less sensitive to the number of GCs. Overall, when the number of GCs is below a limit, sRAT is superior to its equivalent cRAT. However, when the number of GCs passes that limit, cRAT becomes faster. For instance, with 64 architectural and 128 physical registers, having more than x GCs (x >= 20) makes a 6-bit, 128-entry cRAT with x GCs faster than its equivalent 7-bit, 64-entry sRAT. In most cases, cRAT is less energy efficient. To alleviate this energy inefficiency, this works has proposed a physical-level energy optimization for cRAT. Applying the proposed energy optimization to a 6-bit, 128-entry and 6-bit, 512-entry cRATs reduce their lookup energy by 40% and 58% respectively.

5. Compacted Matrix Schedulers

Dynamically-scheduled processors exploit instruction level parallelism by buffering instructions and scheduling them potentially out of the program order. The *instruction scheduler,* responsible for the buffering and scheduling, typically comprises wakeup and select sides. Instructions wait in the wakeup stage to become *ready,* i.e., until all their input operands become available. The select side chooses a set of ready instructions to be sent for execution taking resource constraints into consideration. The scheduling loop formed between the wakeup and the select sides is critical for performance [49]. This scheduling loop's delay is a function of the scheduler size (often synonymous to the window size) and to a lesser extent of the issue width.

The scheduler is a performance-critical component of processors. Large schedulers can improve the instruction per cycle (IPC) completion rate by issuing more instructions every cycle. However, larger schedulers are also slower, and thus can decrease clock period, offsetting any IPC benefits, hence leading to lower overall performance. Accordingly, commercial processors do not use large schedulers (e.g., the Intel and AMD desktop/server processors have integer scheduler sizes of 24 to 32 entries [58]).

The instruction scheduler's wakeup side serves three roles: *(i)* It holds instructions waiting for their inputs to be produced; *(ii)* It matches waiting instructions with incoming results; and, *(iii)* it identifies ready-for-execution instructions (all input operands are available). The matching functionality of the wakeup can be implemented using content addressable memories (CAMs) [49] [25] or dependency matrices [25] [58] [28]. The latter option is faster and consumes less

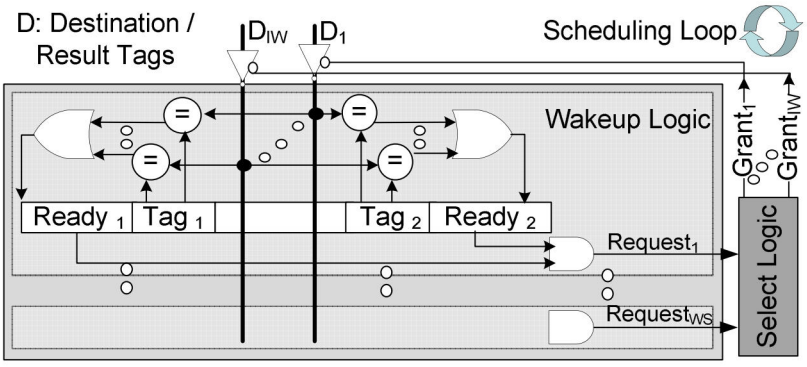


Figure 5-1: CAM-based wakeup

power than the former. The *compacted matrix scheduler* (CMS) exploits typical program behavior to reduce the matrix width, and hence the wakeup latency [58]. In the original matrix scheduler, a column is associated with each instruction; columns serve as communication channels where producers wake up their consumers. In CMS, a column is assigned on demand through an indirection table, called wakeup allocation table or *WAT*.

Although arguments have been made in support of CMS's speed and scalability advantages [58], neither actual physical-level measurements nor models have been reported for it. Models can be used by computer architects to study the latency and energy of various alternatives during the early stages of architectural-level exploration where physical-level implementation is either impossible or impractical to develop due to time and/or cost constraints.

This work investigates the delay and energy variations of CMS as a function of the issue width (IW), the window size (WS), and the number of global checkpoints (NoGCs). For this physical-level investigation, full-custom implementations were developed in a commercial 90 nm fabrication technology. A physical-level study is further useful in exposing design inefficiencies or optimization opportunities. Accordingly, this work proposes an energy optimization that throttles pre-charge and evaluation for unallocated matrix columns. This optimization reduces energy depending on the scheduler size (e.g., for 32×20 and 64×20 matrices, this optimization reduces energy by 10% and 18% respectively). To the best of our knowledge, this is the first work to study the physical-level implementation and optimizations of CMS.

The rest of this chapter is organized as follows: Section 5.1 reviews CAM-based and matrix-based schedulers. Section 5.2 explains the architecture of compacted matrix schedulers or CMS and its accompanying logic, WAT. Section 5.3 and 5.4 discuss physical-level implementation and evaluation results respectively. Finally, Section 5.5 summarizes key findings.

5.1 Background

In typical modern processors, instructions first pass through register renaming, and then enter the instruction scheduler where they wait until they can proceed and execute. The instruction scheduler comprises wakeup and select sides. The wakeup side keeps track of the data

dependencies among WS instructions (without loss of generality, we assume that the window size and the scheduler size are the same). An instruction remains dormant while some of its input/source operands are still outstanding. Once all of the source operands of an instruction become ready, the instruction is "woken up" i.e., becomes ready to send an execution request to the select logic. The wakeup side produces a vector of size WS showing the ready status of all of its instructions. The select side chooses at most IW instructions of those marked as ready in the request vector taking into consideration resource constraints (e.g., availability of functional units). The select side uses a priority encoder that typically chooses the oldest ready instructions.

The rest of this section reviews three scheduler architectures: CAM-based scheduler, regular (uncompressed) matrix-based scheduler, and compacted (compressed) matrix-based scheduler.

5.1.1 CAM-Based Scheduler

In a CAM-based scheduler (e.g., implemented in MIPS R10000), one entry exists per sceduler entry. Every entry includes a tag field and a ready bit per source operand as depicted in Figure 5-1 [49]. As up to IW instructions are selected for execution, their result tags (e.g., indexes of the physical registers or reservation stations) are broadcast on the result buses. Broadcasting of a result tag is delayed for an appropriate number of clock cycles, defined by the instruction's execution latency. Each result bus is connected to comparators, one per source operand tag at each entry. The comparators match the waiting source operand tags against the result tags. For X instructions, each having Y source operands, and with Z result broadcast buses, $X \times Y \times Z \times \text{Log}_2(WS)$ single-bit comparators are required assuming that window size or WS is equal to the number of physical registers. Once all its source operands become ready, an instruction sends an execution request to the select logic.

5.1.2 Matrix-Based Schedulers

The conventional $WS \times WS$ matrix scheduler has a row and a column per instruction [25] [28]. The instruction's column and row indexes are the same as its scheduler entry index. A row records the instruction's input dependencies whereas a column marks the instruction's true output dependencies. The matrix operates as follows. A matrix row c is allocated to an instruction when it enters into the scheduler. For each source operand that is still pending, if the

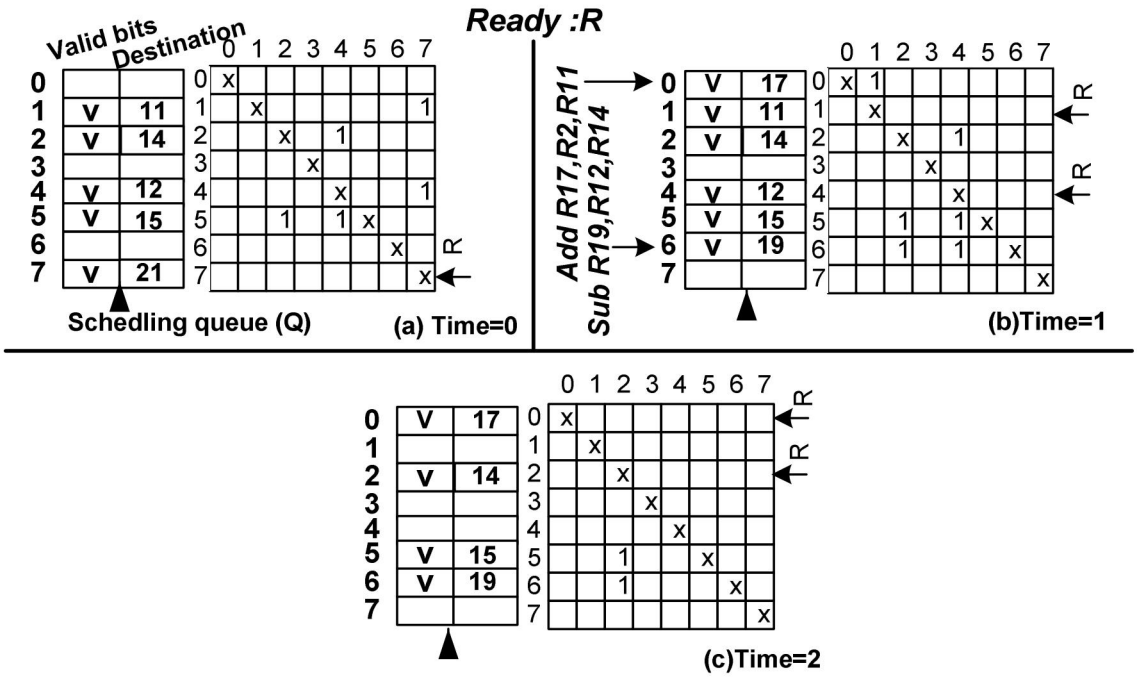
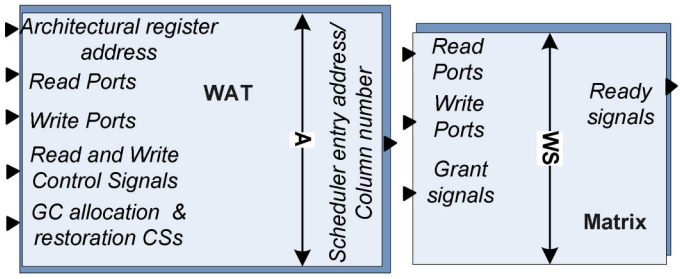


Figure 5-2: *Scheduling with the dependency matrix.*

instruction at the scheduler entry *c* consumes the result produced by the instruction at the scheduler entry *p,* matrix cell (*c*,*p*) is set to one. Just before an instruction produces its result, it clears all cells along its associated column. In this way, the producer notifies all its consumers of its result availability. When a row is all zero, its corresponding instruction turns on its row-ready bit in order to send an execution request to the select logic.

Figure 5-2 (a) to (c) illustrate subsequent snapshots of the wakeup matrix for an eight-entry instruction queue (Q). In addition to a matrix row and a matrix column, a Q entry is associated with an instruction; each Q entry maintains a destination tag, source tags and a valid bit. These fields are used during instruction execution and for clearing the appropriate column. Scheduling decisions are performed using the matrix cells alone. In this discussion, Q_X refers to the instruction stored at scheduler entry x. At time T=0, Q has five valid entries. The cell (1,7) is set since Q_1 depends on Q_7. Q_7 is ready since its row is all clear. In superscalar processors, up to IW instructions can be issued (i.e., sent for execution) per cycle. Also, up to IW instructions can be renamed and allocated into Q. At time T=1, Q_7 issues and accordingly clears its column 7, resulting in Q_1 and Q_4 becoming ready. Also, at time T=1, two new instructions are simultaneously allocated into Q_0 and Q_6 respectively. When new instructions enter into Q, their



Issue width: *IW*
Instruction scheduler/ Window size: *WS*
Number of checkpoints: *NoGC*
Number of source registers per instruction: *S*
Number of destination registers per instruction: *D*
Number of architectural registers: *A*
Number of physical registers: P = WS
Number of columns: *CN =20* *(CN <= Log_2(WS))*

Number of WAT entries: *A*
Number of WAT read ports: *(S+D) x IW*
Number of WAT write ports: *D x IW*
WAT entry width: *log_2 (WS) + 2 + NoGC*

Number of matrix entries: *WS*
Number of matrix read ports: *IW*
Number of matrix write ports: *IW*
Matrix entry width: *CN*

Figure 5-3: Geometries and block diagrams for the CMS and the WAT

dependencies are recorded in the matrix. Q_0 uses R_{11}, produced by Q_1, hence cell (0,1) is set. Q_0 also uses R_2, which is ready. The cells (6,2) and (6,4) are set for Q_6 as it uses R_{14} and R_{12} (produced by Q_2 and Q_4) respectively. At time T=2, Q_1 and Q_4 become ready; hence, their columns and scheduler entries are cleared. As a result, Q_0 becomes ready. However, Q_5 and Q_6 are still waiting for Q_2 producing R_{14}.

5.2 Compacted Matrix Scheduler (CMS)

The size of the conventional matrix is proportional to WS^2 making larger schedulers difficult to implement. However, in practice, similar IPC performance can be achieved using a compacted matrix with fewer columns [58]. Several observations motivated compacted matrix schedulers: *(i)* Irrespective of the scheduler size, about 70% of the instructions either have no consumer(s) in the scheduler or do not need to broadcast any result (e.g., branch or store); *(ii)* Large schedulers are rarely full of producers and are often still refilling from pipeline flushes; and, *(iii)* Matrix

snapshots during execution show that a few active dependencies exist per cycle, thus most of the WS matrix columns are unused at any given time, and hence the matrix width can be reduced. Previous work demonstrates that 12-16 columns or up to 20 columns are sufficient for schedulers with up to 128 or more entries respectively [58].

CMS takes advantage of the aforementioned observations and postpones allocating a column to an instruction until its first consumer enters the scheduler. Reducing the matrix width improves the speed of larger schedulers at the cost of a negligible IPC performance loss [58]. CMS uses a redirection table, called wakeup allocation table or WAT, to assign a column to a producer instruction. The WAT is required because in the compacted matrix unlike the conventional matrix, the instruction's column index is not the same as its scheduler entry index. To assign a column to a producer, WAT assigns the column to its destination register. The WAT manages column subscriptions by mapping consumer's source registers to producer's column indexes. A column free list (CFL) keeps a list of free columns.

5.2.1 Wakeup Allocation Table (WAT)

Shown in Figure 5-3, the WAT has one entry per architectural register. Each WAT entry can be in one of the following four states: *unallocated* (the register value is unavailable, and the producer has not been assigned a column yet), *allocated* (the producer has been assigned a column, but its result, the register value, is not ready yet), *ready* (the register value is available), and *de-allocated* (the column was released after the producer and/or all of its consumers left the scheduler). The WAT entry width is $Log_2(WS)+2$ as it stores either a matrix row entry (equal to the scheduler entry) index or a matrix column index and two bits to encode one of the four WAT states. The WAT operations are lookup (read), update (write), GC allocation and GC restoration.

The rest of this section describes WAT operations and WAT outputs by means of an example. Using the instruction sequence shown in Figure 5-4, the rest of this section discusses the following scenarios: a producer enters the scheduler (*CASE 1*); the producer's first consumer enters the scheduler (*CASE 2*); the producer's subsequent consumer enters the scheduler (*CASE 3*); the producer executes (*CASE 4*); and, the producer's associated column is released (*CASE 5*).

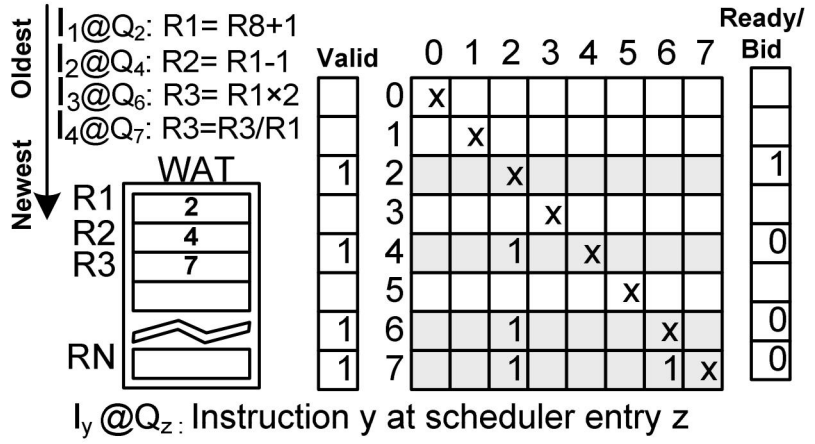


Figure 5-4: Instruction sequence example

CASE 1 – Producer enters the scheduler (WAT Update): When a new instruction enters the scheduler, the WAT entry corresponding to the instruction's destination architectural register is updated with its row index (or scheduler entry index); however, no column is allocated for the instruction. For example, when I_1 with destination R_1 enters Q_2, the WAT entry associated with R_1 is set to "unallocated, 2", meaning that R_1's value will be produced by I_1 at Q_2.

In superscalar processors, up to IW scheduler entries may be concurrently allocated. Hence, up to IW WAT updates may proceed simultaneously. However, when write-after-write (WAW) dependencies exist, only the last update should proceed into the WAT. Hence, WAW dependencies must be detected among the IW co-renamed instructions; the destination of each instruction is compared with those of its preceding instructions. The WAT update is restricted to the last writer for a specific destination architectural register (I_4 at Q_7 updates the WAT entry for the destination R_3).

CASE 2 – First consumer enters the scheduler (a WAT Lookup followed by a WAT Update): Assume that I_2 with source R_1 is allocated into Q_4. The WAT lookup for source R_1 returns "unallocated, 2", indicating that R_1 is produced by I_1 at Q_2. Since I_2 is the first consumer of I_1, a free column (e.g., 3) is allocated for I_1 by the CFL. The WAT entry for R1 is set to "allocated, 3", meaning that as of now column 3 belongs to I_1, the producer of the architectural register R_1. I_1 must also be informed of its newly-assigned column index so that it can clear its column when its result becomes ready. An extra field per scheduler entry keeps the column index if any

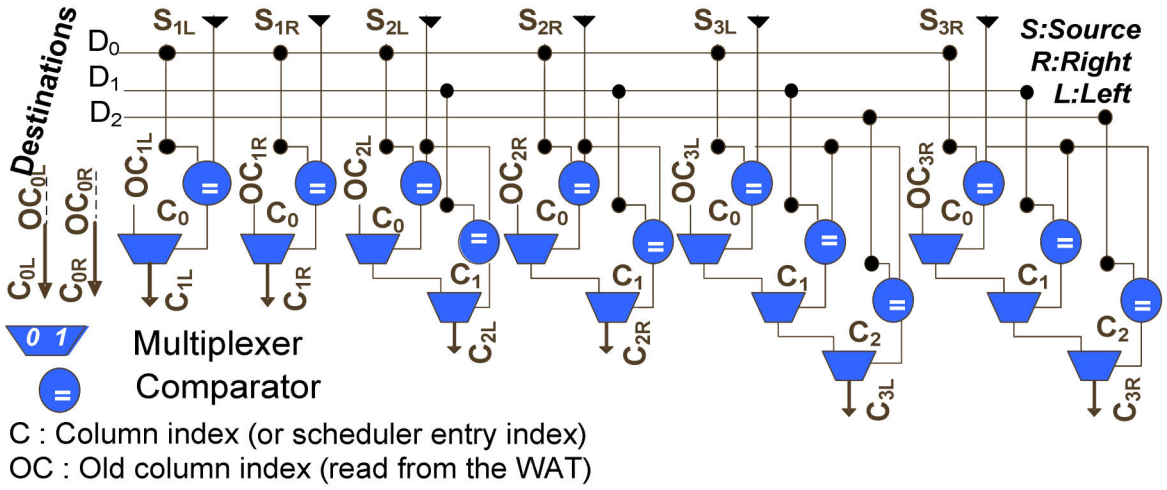


Figure 5-5: RAW dependency checking

column is allocated. Finally, the matrix cell (4,3) is set to one to record the read-after-write (RAW) dependency of I_2 on I_1.

CASE 3 – Subsequent consumer enters the scheduler (WAT Lookup): I_3 at Q_6 is a subsequent consumer of R_1. The WAT lookup for source R_1 returns "allocated,3", and the matrix cell (6,3) is set to record the RAW dependency of I_3 on I_1. This and the preceding scenarios show that a WAT lookup for a source architectural register returns either the producer's row index (*CASE 2*) or the producer's column index (*CASE 3*).

In superscalar processors, if RAW dependencies exist between a pair of IW co-renamed instructions, the producer's columns index (e.g., I_1) may be assigned at the same cycle a consumer needs it (e.g., I_2 or I_3). However, the WAT will not be updated by the end of the cycle. Hence, RAW dependencies among co-renamed instructions must be detected. As depicted in Figure 5-5, to detect RAW dependencies, the sources of each instruction are compared against the destinations of all its co-renamed preceding instructions. The circuitry for detecting RAW and WAW dependencies can be shared between the renaming' register alias table (RAT) and the WAT because renaming also requires this information.

Renaming/Dispatch	Scheduling	Execution
WAT lookup for sources/ WAT lookup for destinations/ Matrix update	Matrix read Producing row-ready bits	Matrix column clearance/ WAT update for destination

Figure 5-6: Actions taken by CMS and WAT at various execution stages

CASE 4 – Producer completes execution (WAT Update): When producer I_1 starts execution, after an appropriate number of cycles, depending on I_1's execution latency, its column (column 3 has been assigned to I_1) is a cleared, and the matrix-based wakeup proceeds (Section 5.2.3). At this point, the R1's WAT state changes to *read*y. If the producer needs to clear its column at the same time a consumer enters the scheduler, the column reset will be prioritized over the dependency-induced matrix cell set. Otherwise, deadlock can occur as the consumer would await a never-happening column reset.

CASE 5 – Column is released (WAT Update): A column is freed if *(i)* its producer and/or *(ii)* all its consumers have left the scheduler. The latter is an optimization for when a pipeline flush occurs between a producer and its consumer(s). Even if the producer stays in the scheduler, the register state can be changed to de-allocated and its column can be freed. If the processor supports speculative scheduling (i.e., consumers of a load are granted execution predicting a hit in the L1 data cache), a column is not freed once the producer writes back its result. If the load misses, the consumers will be reset in the scheduler and will wait for a second column broadcast from the load upon hit. Thus, an instruction can broadcast on the same column multiple times. In this case, only after the producer leaves the scheduler, its associated column will be freed.

5.2.2 Recovery from Mispeculation

The WAT like the RAT is a speculative structure and requires support to recover from mispeculations. For WAT, recovery can be done using GCs; GCs are copies of the WAT content taken when a rollback due to mispeculation is probable (e.g., on predicted branches).

Checkpoints for the matrix are not required if the valid bits for the scheduler entries and matrix rows of the squashed instructions are cleared. When pipeline flushes occur, some rows may keep obsolete information, which does no harm if the select logic is prevented from receiving false requests. To do so, for each matrix row, the valid bit is ANDed with the ready bit to produce the execution request to the select logic.

5.2.3 Dependency Matrix Operations

Shown in Figure 5-3, the compacted matrix has WS rows and 20 columns (Section 5.2). The

matrix operates as follows. A matrix row is *updated* for a new instruction *c* by finding the column indexes for the producers of the instruction's source operands from the WAT (e.g., p_1 and p_2) and setting the matrix cells (e.g., (c,p_1) and (c,p_2)) to one. These steps proceed in parallel with rename and dispatch (writing the instruction into the scheduler). Before an instruction completes execution, its matrix *column* is *cleared*. When the instruction's matrix row cells are all zero, its row-ready bit is *set*, interpreted as a request by the select logic. Figure 5-6 summarizes the actions that take place in the CMS and WAT.

5.3 Physical-Level Implementation

This section describes the physical-level implementation of the WAT and CMS.

5.3.1 WAT

A non-checkpointed WAT is a multi-ported register file. Assuming a MIPS-like instruction set architecture, where the instructions may have at most two source operands and one destination operand, the WAT needs to support 3xN reads and N writes per cycle, where N is the number of instructions required to be renamed per cycle. 2xN read ports are used to rename the two source operands, and another N read ports are needed to read the current mappings of the destinations for the purpose of recovery using the reorder buffer. N write ports are also used to write new mappings for the destinations.

Depicted in Figure 5-3, a checkpointed WAT embeds GCs, copies of the WAT cells. Figure 5-8(a), shows the main WAT cell with multiple read and write ports. The GC cells, depicted in Figure 5-8(b), are organized in a bi-directional shift register, shown in Figure 5-8(c). Figure 5-8(d), (e) and (f) respectively illustrate the precharge circuitry, the sense amplifier logic used during reads, and the write drivers used during writes.

This work uses the serial-access-buffer (SAB) GC organization mechanism as it is faster and simpler than the alternative. SAB organizes the GCs in a bi-directional shift register with connections between adjacent cells; only one of the GCs is connected to the main WAT cell through pass gates [26]. GC allocation is done by shifting the GC cells to the right, copying the WAT data cell to the adjacent vacant position. Restoring from a GC may require multiple steps

since the appropriate GC must be shifted into the WAT main cell. A SAB GC cell consists of a register and a multiplexer controlling the shift direction. SAB uses two non-overlapping clocks and two external control signals irrespective of the number of GCs. Multiple reads may access the same WAT entry; hence, the main WAT cell must be capable of driving a capacitance proportional to the number of WAT ports and GC(s). To protect the WAT cell's data during multiple accesses, decoupling buffers isolate the WAT data cell and the read ports [74] [26]. Differential read and write operations are used due to their superior latency, energy and noise margins. To reduce power, the following techniques were employed: *(i)* pulse operations for the wordlines, periphery circuits and sense amplifiers; *(ii)* multi-stage static CMOS decoding; and, *(iii)* current-mode read and write operations.

5.3.2 Compacted Matrix

Figure 5-7(a) shows the architecture of the compacted matrix scheduler and its interface with the select logic. Figure 5-7(b) also illustrates the matrix column's transistor-level implementation. The matrix receives its inputs, the grant (or upcoming execution completion) signals, from the select logic, and produces the row-ready outputs (the data inputs are active- low). When a new instruction is allocated into a scheduler entry, its associated matrix row is updated with a bit pattern where the dependencies for instruction sources are set by the data decoders. Up to IW instructions may enter the scheduler concurrently; hence, up to IW matrix rows may be updated simultaneously. Therefore, the matrix includes IW data decoders that provide IW sets of bit patterns, each with as many bits as the number of matrix columns (20 in our implementation as per discussion of Section 5.2).

Each matrix cell consists of an SRAM cell. The write ports for updating rows, read/write ports for clearing columns and read ports for producing row-ready outputs are single-ended. Up to IW instructions may write into IW different matrix rows simultaneously. Hence, each matrix cell needs IW write ports. Figure 5-7(b) shows the transistor-level implementation of a matrix column. The grant signals entering from the right side become vertical and are sent over the corresponding columns. The vertical *broadcast* signals clear the corresponding columns. A bitwise-NOR of all matrix cells along a row generates the row's row-ready output (single-ended read ports implement the NOR gate). Up to IW different columns may be cleared

simultaneously. The row-ready signals are first pre-charged to high. A single *set* matrix cell per row (in this implementation, a set cell keeps the value zero) can pull-down the row-ready signal to zero. If all row cells are *unset* (i.e., all cells are set to one), the row-ready stays high, interpreted as a request for execution by the select logic. Producing all row-ready output signals proceeds in parallel.

For column subscription, an additional SRAM cell per column is set (to zero) to indicate that the column is currently allocated. Up to IW columns may be allocated per cycle during renaming; hence, each *column-subscription* cell, which exists per matrix column, needs IW read/write ports. These ports are also used for column de-allocation (by setting the column-subscription cell to one). Up to IW columns may need to be cleared during result write-back. For each column, the column-subscription cell data is ANDed with the input grant to determine if broadcast on the column is permitted. In compacted matrix, compared to a conventional matrix, the row-ready critical path has one extra gate delay penalty due to the column-subscription technique; however, the compacted matrix's reduced latency is supposed to offset this penalty.

If the grant signal for a column is active, the normal precharge and evaluation operations

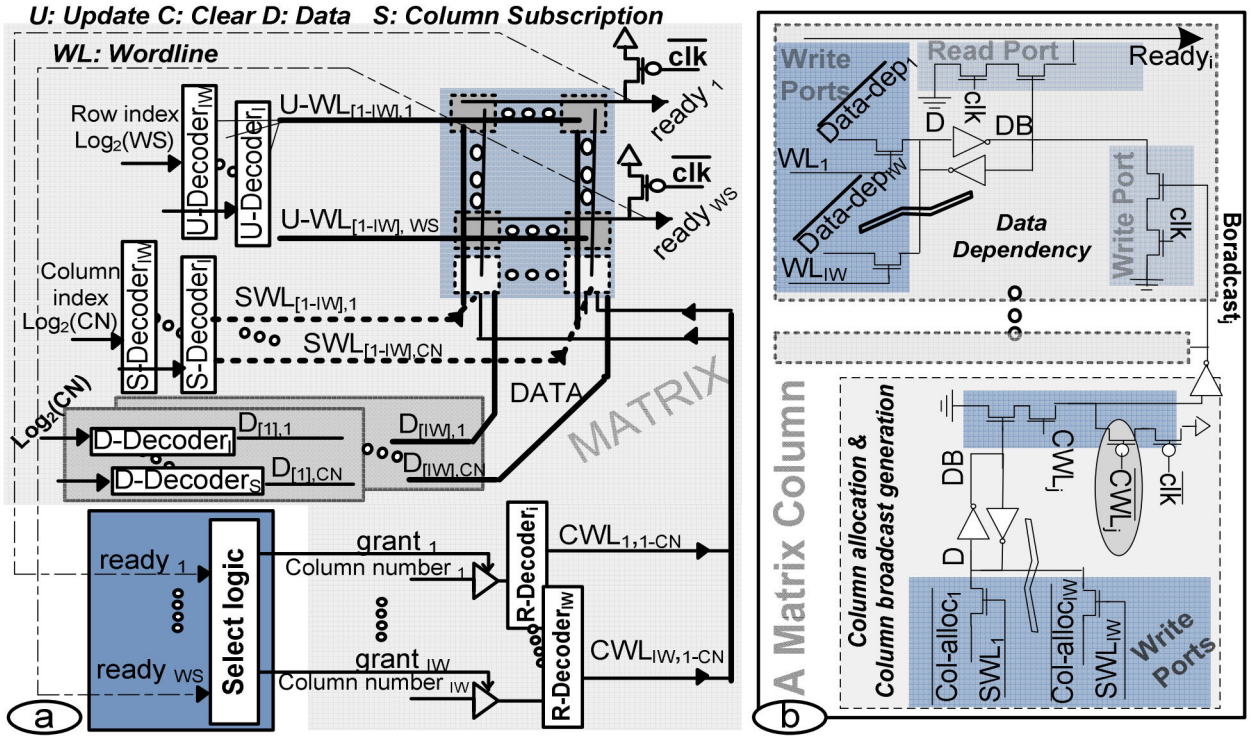


Figure 5-7: Matrix (a) overcall organization and (b) column's transistor-level implementation

proceed; otherwise, these operations are unnecessary. To prevent unnecessary activities, and hence to reduce the energy consumption, an extra switch (pass transistor) per column, shown circled in Figure 5-7(b), is added. For 32×20 and 64×20 matrices, this optimization reduces total energy by 10% and 18% respectively.

5.4 Evaluation

This section presents the latency and energy measurements from the physical implementation for CMS and WAT. Section 5.4.1 details the implementation and measurement methodologies. Sections 5.4.2 and 5.4.3 report the latency and energy measurements and analysis for the CMS and WAT respectively. Finally, Section 5.4.4 presents empirical models for quick estimation of delay and energy.

5.4.1 Design and Measurement Methodology

We restrict our attention to WATs with 0, 4, 8 or 16 GCs because previous work demonstrated that for the SRAM-based RATs, 16 GCs or less are sufficient to achieve performance close to what is possible with infinite GCs [46]. WAT GC count would affect performance exactly as RAT GC count would. The number of architectural registers is either 32 or 64 corresponding to

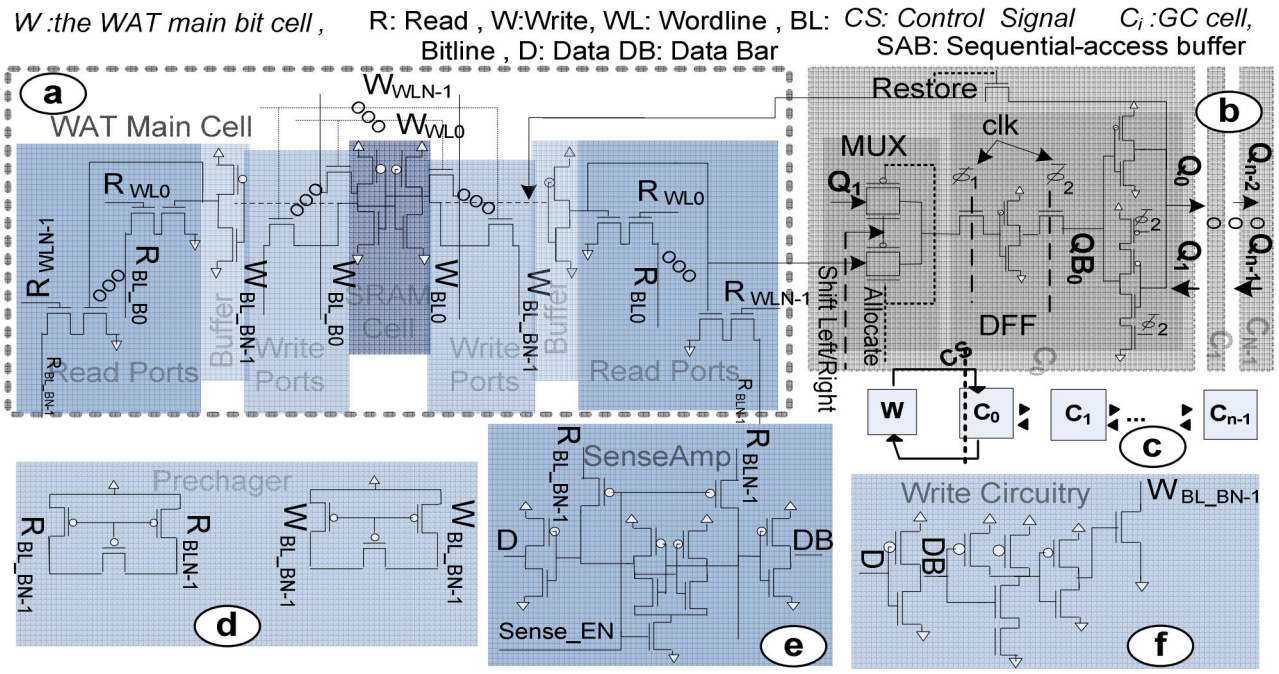


Figure 5-8: WAT building blocks (a) SRAM cell, (b) SAB GC cell, (c) SAB GC organization mechanism, (d) pre-chargers, (e) sense amplifiers and (f) write drivers

typical processors without or with floating-point register sets. The number of physical registers varies from 32 to 256 in power of two steps. We developed full-custom layouts for all components using the Cadence tool set in a 90 nm commercial fabrication technology with a 1.2V supply voltage. For circuit simulations, Spectre™ was used, and worst-case latency and energy values were reported.

Circuit designs can be tailored to achieve different latency and energy tradeoffs. In an actual commercial design, a target latency and/or energy is decided and used as a specification for tuning components. In lieu of an actual specification for the target operating frequency, we used CACTI 4.2 [65], a tool providing latency, power and area estimations for caches and SRAMs, to obtain upper bounds on the latencies of the SRAMs that would be similar in size to the WAT. Using CACTI, for the base 4-way superscalar WAT, we determined an upper bound on the critical path delay by estimating the delay of a 64-bit, 64-entry SRAM with 12 read and four write ports. This upper bound is reasonable since non-checkpointed WATs/RATs are identical to multi-ported SRAMs.

5.4.2 Compacted Matrix

Figure 5-9 (a) and (b) report compacted matrix read and write latencies for 2-way and 4-way superscalar schedulers respectively. Figure 5-10 shows the corresponding energy measurements. The number of rows (equal to WS) varies from 32 to 512. Each matrix has 20 columns (Section 5.2).

The matrix read delay (wakeup delay) is the time between the select logic's grant activation and row-ready discharge. Increasing the number of matrix rows increases matrix read latency as it increases the size, and hence the latencies of the decoder and of the select logic's priority encoder. For example, when WS increases eight-fold from 64 to 512, read latency increases by 46.7%. The scheduler delay is the sum of the matrix read delay (or wakeup delay) and the select logic delay. The scheduler (wakeup plus select) delay increases logarithmically with the number of matrix rows (equal to WS) when IW is fixed. The matrix write delay is the time taken to update the matrix cell values. Matrix read latency is longer that matrix write latency. For a fixed IW, matrix write delay and energy increase logarithmically with the number of rows.

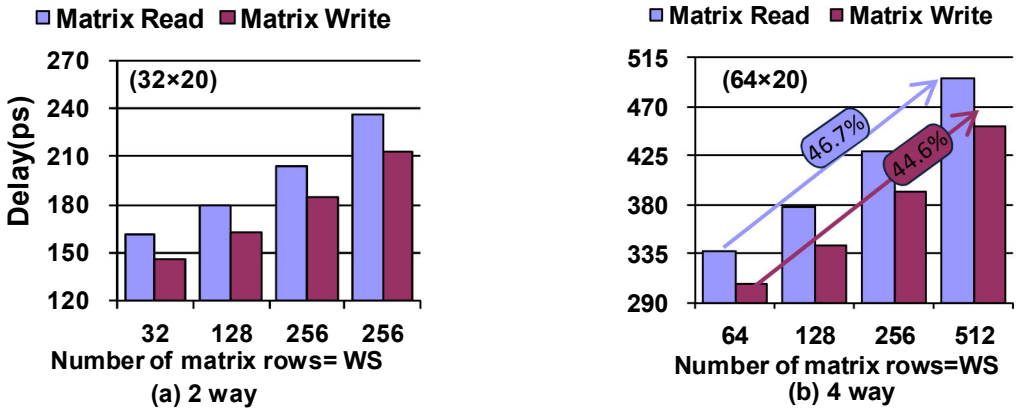


Figure 5-9: Matrix read delay and matrix write delay for (a) 2-way and (b) 4-way schedulers.

Increasing the number of matrix columns (matrix width) increases row-ready NOR gate's size, and hence the wakeup delay. Increasing the matrix width also increases the load on the row clock driver, and hence the driver's delay. By transistor resizing and output buffering, significant delay increase can be avoided at the cost of more energy. Although we do not show these results in the interest of space, for a fixed WS and IW, both read and write latencies increase linearly with a very small slope for larger matrix widths. For example, for the 64×20 matrix, the read delay increases by 21.6% as the number of column increases from 16 to 20.

Previous work demonstrates the speed and power efficiency of the conventional (uncompressed) matrix scheduler compared to the CAM-based scheduler [25]. The same conclusions are applicable to the compacted matrix schedulers.

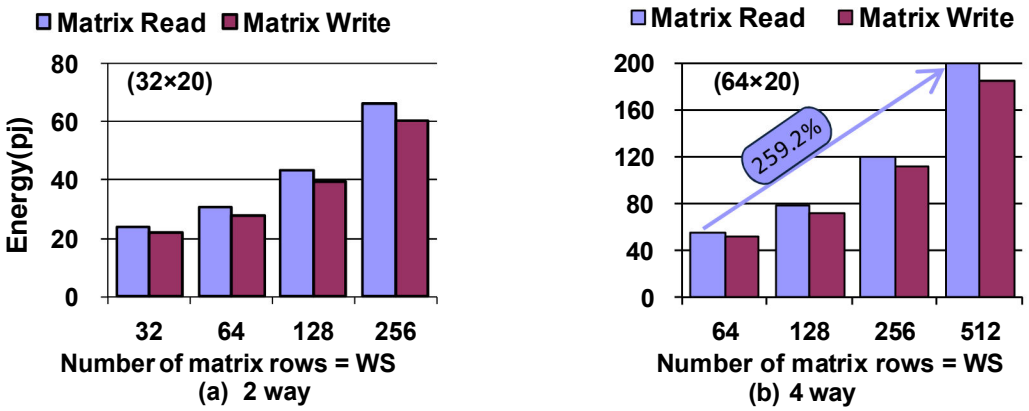


Figure 5-10: Matrix read energy and matrix write energy for (a)2-way and (b) 4-way schedulers.

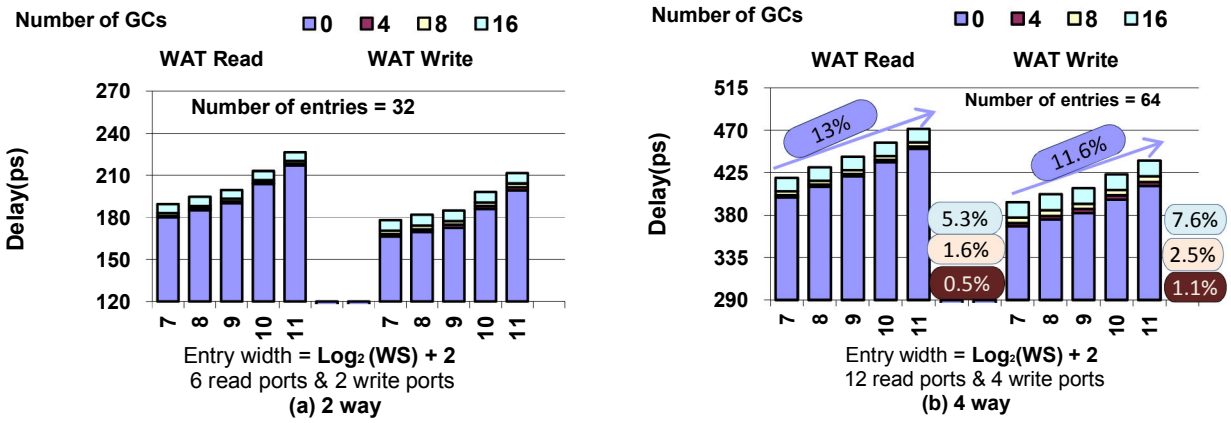


Figure 5-11: WAT read and write latency for (a) 2-way and (b) 4-way schedulers

5.4.3 WAT

Figure 5-11 reports WAT read and write latency as the entry width varies from 7 to 11 (corresponding to schedulers of 32 to 512 entries). For each entry width, four stacked bars are presented each for WATs with 0, 4, 8, and 16 GCs.

For a fixed IW and NoGCs, WAT latency increases logarithmically with the entry width. For example, WAT read and write latencies increase by 13% and 11.6% as the entry width increases from 8 to 11 (WS increases from 64 to 512) for the 4-way WAT with no GCs.

For a fixed WS and IW, both latency and energy increase exponentially with increasing NoGCs. For example, for the 4-way WATs, writes are 1.1%, 2.5% and 6.7% slower with 4, 8 and 16 GCs compared to the non-checkpointed WAT. Previous work shows that a RAT using very few GCs (e.g., four) leads to optimal overall performance (execution time) because of the impact of increasing NoGCs on renaming latency [6]. Our results show a similar trend for the WAT.

Figure 5-12 shows the WAT energy as a function of the entry width ($log_2(WS)+2$) and NoGCs. Energy increases slightly by increasing the entry width due to the additional bitlines and sense amplifiers, and longer wordlines. The GC allocation's and restoration's latency and energy are not shown in the interest of space. The measurements for these operations are negligible compared to that of the WAT reads and writes.

A comparison of the 2-way and 4-way delay measurements for the CMS and WAT suggests that

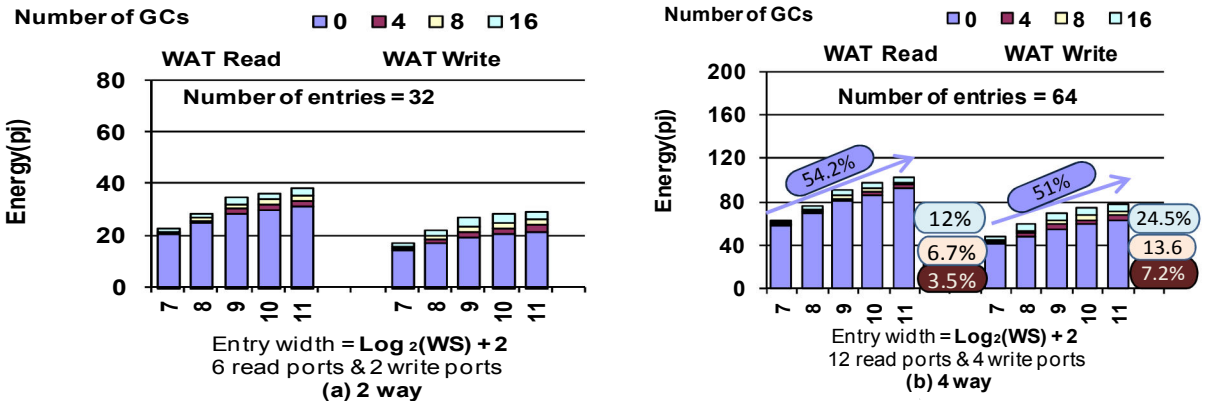


Figure 5-12: WAT read and write energy for (a) 2-way and (b) 4-way schedulers.

latency increases quadratically as IW increases. Increasing IW increases the number of read and write ports for both CMS and WAT; hence, it significantly increases latency and energy.

5.4.4 Simplified, Empirical Models

We also developed empirical delay and energy models for the CMS and WAT. Computer architects can use these models when various design options are considered during architectural-level exploration where developing physical-level implementation is impractical. To develop the empirical models, we applied curved fitting over 60 data points, based on the measurements from our physical-level implementations in 90 nm technology [8]. In the interest of space, we only present the latency models for the 4-way matrix and WAT. Equations (76) and (77) estimate matrix read and write latencies, while equations (78) and (79) estimate WAT read and write latencies. The worst-case relative error of the model predictions is within 7.8% of the Spectre™ circuit simulation results for the data points we used for curve fitting (0<=WS<=512, 0<=NoGCs<=16, IW=2 or 4). These data points cover values of the WS, IW and NoGCs that would be typical for modern processors, and model predictions have a rage of accuracy that is acceptable for architectural-level studies. For predicting the delay and energy of the configurations outside of this range, extrapolation can be used, which is the process of constructing new data points outside a discrete set of known data points. However, the extrapolation results are often subject to greater uncertainty.

$$\text{Matrix}_{\text{Read Delay}}\,(\text{ps}) = 155.38 \times WS^{0.1846} \tag{76}$$

$$\text{Matrix}_{\text{Write Delay}}\,(\text{ps}) = 58.653 \times \ln(WS) + 69.556 \tag{77}$$

$$WAT_{Read\ Delay}(ps) = 1.7543 \times e^{0.1524 \times NoGCs} + 22.77 \times \ln(WS) + 330.42 \qquad NoGCs >= 0 \qquad (78)$$

$$WAT_{Write\ Delay}(ps) = 5.5818 \times e^{0.1322 \times NoGCs} + 13.946 \times \ln(WS) + 330.42 \qquad NoGCs >= 0 \qquad (79)$$

5.5 Conclusion

This work investigates the delay and energy variations of the recently proposed compacted matrix scheduler (CMS) and its accompanying logic, wakeup allocation table (WAT). Previous work discussed the speed and scalability advantages of CMS; however, neither actual physical-level investigations nor models were reported for it. Using full-custom layouts in a commercial 90 nm fabrication technology, this work studies the latency and energy variation trends of CMS and WAT as a function of the issue width, the window size and the number of global checkpoints. An energy optimization for the matrix has also been proposed that reduces energy by 10% or 18% depending on the scheduler size.

Our results show that for a fixed issue width and global checkpoint count, CMS delay and energy increase logarithmically with increasing the number of matrix rows (or the window size). For a fixed window size, issue width and global checkpoint count, the matrix delay increases linearly with increasing the number of columns. For a fixed issue width and global checkpoint count, WAT latency increases logarithmically for larger window sizes. For a fixed window size and issue width, both WAT latency and energy increase exponentially with increasing checkpoint count. The results of this work support the previously claimed latency and energy scalability of the compacted matrix schedulers.

6. Conclusion

Computer architects consider a myriad of possible organizations and designs to decide which best meets the constraints on performance, power and cost for each particular processor. This thesis was motivated by the need of computer architects to gain insight into the physical-level characteristics of various designs evaluated during early stages of the design process.

Actual measurements of delay, power and area for a design will only be available after the design is implemented at the physical level. However, implementing all the designs considered during architectural-level exploration is impossible or unaffordable due to time and/or cost constraints. Analytical and empirical models can provide early estimations of the physical-level characteristics of the designs in lieu of actual measurements. Existing models tend to be out-dated for three reasons: *(i)* They have been developed based on old circuits in old fabrication technologies; *(ii)* The high-level designs of these components have evolved and older designs may no longer be representative; and, *(iii)* The overall architecture of processors has changed significantly, and new components for which no models exist have been introduced or are being considered. Thus, existing models cannot provide reliable estimations of speed, power and area for designing future processors implemented in advanced fabrication technologies.

Considering the aforementioned shortcomings in the modeling and optimization areas, this thesis *(i)* investigated the hardware complexity characteristics (speed, power and area) for several key components of modern high-performance processors, *(ii)* developed latency and energy analytical and/or empirical models for these components, and *(iii)* proposed architectural- and/or physical-level optimizations for these components. Specifically, this thesis studied three components of modern dynamically-scheduled, superscalar processors: Counting Bloom Filters (CBFs), Checkpointed Register Alias Tables (RATs), and Compacted Matrix Schedulers (CMSs). CBFs optimize membership tests and are often used in predictors which in turn indirectly increase the concurrency of processing within a processor (e.g., cache miss predictors); RAT is the core of the renaming stage, which eliminates unnecessary data dependencies across instructions, and thus exposes more ILP; CMS is an implementation of the instruction scheduler, the main component exploiting ILP. No comprehensive physical-level study or models have been

reported for these components. This thesis *(i)* proposed a novel physical-level implementation for CBFs, *(ii)* studied the effect of the number of RAT checkpoints, and *(iii)* investigated CMS at the physical-level and proposed an energy reduction technique. Based on the physical-level investigations of these components, this thesis developed simple, empirical latency and energy models as well as detailed, analytical models adaptable for newer fabrication technologies.

This thesis reviewed how hardware CBFs help improve the speed and energy of membership tests by maintaining an imprecise and compact representation of a large set to be searched. This thesis investigated the latency and energy of CBF implementations using full-custom layouts in a 0.13 μm fabrication technology. The previously-assumed implementation, S-CBF, uses an SRAM array of counts and a shared up/down counter. Our novel implementation, L-CBF, utilizes an array of up/down linear feedback shift registers and local zero detectors. Circuit simulations showed that for a 1K-entry CBF with a 15-bit count per entry, L-CBF compared to S-CBF is 3.7x or 1.6x faster and requires 2.3x or 1.4x less energy depending on the operation. The thesis also presented analytical energy and delay models for L-CBF. The model estimations are within 5% and 10% of Spectre™ simulation results for latency and energy respectively.

This work also investigated how the latency and energy of the RAT varies as a function of the number of GCs, the issue width, and the window size. Previous work on RAT checkpointing focused solely on evaluating performance in terms of IPC. This thesis complements the previous work by taking into consideration the impact of these changes on the clock cycle as well. This thesis studied the impact of the number of checkpoints on actual performance (execution time) relying on full-custom checkpointed RAT implementations developed in a commercial 0.13 μm fabrication technology. This work showed that, as expected, focusing on IPC alone incorrectly predicts performance. The results of this thesis justify those checkpointing techniques that aimed at reducing the number of checkpoints as much as possible. Additionally, this work studied two different GC management mechanisms, RAB and SAB. Although RAB, representative of recent checkpointed RAT designs, offers better IPC performance, SAB offers better actual performance (execution time). This work also presented empirical and analytical latency and energy models for checkpointed SRAM-based RATs. The analytical latency and energy models offer estimations within 6.4% and 11.6% of circuit simulation results respectively.

This work also determined and compared the latency and energy variation trends of the checkpointed SRAM-based RAT (sRAT) and checkpointed CAM-based RAT (cRAT) as a function of the window size, the issue width, and the GC count. Compared to sRAT, cRAT is more sensitive to the number of physical registers and issue width, and less sensitive to the GC count. Results show that when the number of GCs exceeds a limit, cRAT becomes faster than its equivalent sRAT. For instance, with 64 architectural registers and 128 physical registers, having more than 20 GCs makes a 6-bit, 128-entry cRAT faster than its equivalent 7-bit, 64-entry sRAT. In most cases, cRAT is less energy efficient than sRAT, hence this work proposed an energy optimization for the cRAT that selectively disables cRAT entries that do not result in a match during lookup. For a 128-entry cRAT, energy is reduced by 40% by this optimization.

Previous work focused on the architecture of CMS and argued in support of its speed and scalability advantages. However, neither physical-level investigations nor analytical models have been reported for this compacted matrix scheduler. Using full-custom layouts in a commercial 90 nm fabrication technology, this work investigated the latency and energy variations of the matrix and its accompanying logic as a function of the issue width, the window size and the number of GCs. This work also proposed an energy optimization for CMS. Specifically, for 32×20 and 64×20 matrices, this optimization reduces total energy by 10% and 18% respectively.

6.1 Future work

Existing models need to be updated to take into consideration new circuit design practices and deep sub-micron fabrication technology challenges (e.g., leakage current and interconnect delay). Hence, developing new models or refining available models for other processor components can be the focus of future work. Examples of the candidate components are as follows: scheduler (wakeup and select logic), register files, data bypass logic, reorder buffer, load/store queue, caches and various predictors. Future modeling efforts can also be focused on developing models for various implementations for a specific component (e.g., RAT can be implemented using SRAM or CAM structures). Developing models that let us choose various circuit design styles for the building blocks of a component can also be the goal of future work (e.g., static or dynamic logic, single-ended or differential read or write for bitlines in register files/caches).

References

[1] B. Agrawal, T. Sherwood, "Guiding Architectural SRAM Models", International Conference on Computer Design, 376-382, Oct 2007

[2] B. Agrawal, T. Sherwood, "Ternary CAM Power and Delay Model: Extensions and Uses", IEEE Transaction on VLSI, 16(5): 554-564, May 2008.

[3] H. Akkary, R. Rajwar and S. Srinivasan, "An Analysis of Resource-Efficient Checkpoint Architecture", ACM Transactions on Architecture and Code Optimization, 1(4):418-444, Dec. 2004.

[4] H. Akkary, R. Rajwar and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Instruction Window Processors", IEEE/ACM International Symposium on Microarchitecture, 423-434, Dec. 2003.

[5] P. Akl and A. Moshovos, "BranchTap: Improving Performance with Very Few Checkpoints through Adaptive Speculation Control", International Conference on Supercomputing, 36-45, Jun. 2006.

[6] P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators", Xilinx, Application Note 052, Jul. 1996.

[7] B. S. Amrutur and M. A. Horowitz, "Fast Low-Power Decoders for RAMs", IEEE Journal of Solid-State Circuits, 36(10):1506- 1515, Oct. 2001.

[8] B. S. Amrutur, "Design and Analysis of Fast Low Power SRAMs", Ph.D. Dissertation, Electrical Engineering Department, Stanford University, 1999.

[9] B. S. Amrutur and M. A. Horowitz, "Speed and Power Scaling of SRAM", IEEE Journal of Solid-State Circuits, 35(2):175-185, Feb. 2000.

[10] P. H. Bardell, W. H. McAnney, and J. Savir, Built-in Test for VLSI: Pseudo-random Techniques, John Wiley & Sons Inc., 1987.

[11] B. Bishop, T. P. Kelliher and M. J. Irwin, "The Design of A Register Renaming Unit", Great Lakes Symposium on VLSI, 34-37, Mar. 1999.

[12] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural Level Power Analysis and Optimizations", International Symposium on Computer Architecture, 83-94, Jun.2000.

[13] D. Brooks, J.D. Wellman, P. Bose and M. Martonosi, "Power-Performance Modeling and Tradeoff Analysis for a High-End Microprocessor", Power Aware Computing Systems Workshop at ASPLOS-IX, 126-136, Nov. 2000.

References

[14] D.M Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta and P.W. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors", IEEE Micro, 20(6): 26-44, Nov.2000.

[15] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, P. G. Emma and M. G. Rosenfield, "New Methodology for Early-stage, Microarchitecture-level Power–Performance Analysis of Microprocessors", IBM Journal of Research and Development, 47(5/6):653-670, Nov. 2003.

[16] D. Burger and T. Austin. "The Simplescalar Tool Set v2.0", Technical Report UW-CS-97-1342, Computer Sciences Department, University of Wisconsin-Madison, Jun. 1997.

[17] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky and B. Williams, "Organization and Implementation of the Register-renaming Mapper for Out-of-Order IBM POWER4 Processors", IBM Journal of Research and Development, 49(1):167-188, Jan. 2005

[18] X.N. Chen and L.S. Peh, "Leakage Power Modeling and Optimization of Interconnection Network", International Symposium on Low Power Electronics and Design, 90-95, Aug. 2003.

[19] S. L. Coumeri and D. E. Thomas, "Memory Modeling for System Synthesis", International Symposium on Low Power Electronics and Design, 179-184, 1998.

[20] A. Cristal, D. Ortega, J. Llosa and M. Valero, "Kilo-Instruction Processors", International Symposium on High Performance Computing, 10-25, Oct. 2003.

[21] A. De Gloria and M. Olivieri , "An Application Specific Multi-Port RAM Cell Circuit for Register Renaming Units in High Speed Microprocessors", IEEE International Symposium on Circuits and Systems, 934-937, May 2001.

[22] M. Q. Do, M. Drazdziulis, P. Larsson-Edefors and L. Bengtsson, "Parameterizable Architecture-Level SRAM Power Model using Circuit Simulation Backend for Leakage Calibration", International Symposium on Quality Electronic Design, 557-563, Mar. 2006.

[23] R. Evans and P. Franzon, "Energy Consumption Modeling and Optimization for SRAMs", IEEE Journal of Solid-State Circuits, 30:571-579, May 1995.

[24] O. Ergin, K. Ghose, G. Küçük and D. Ponomarev, "A Circuit-Level Implementation of Fast, Energy-efficient CMOS Comparators for High-Performance Microprocessors", International Conference on Computer Design, 118-121, Sept. 2002.

References

[25] M. Goshima , K. Nishino , T. Kitamura, Y. Nakashima, S. Tomita and S. Mori, “A High-speed Dynamic Instruction Scheduling Scheme for Superscalar Processors”, ACM/IEEE International Symposium on Microarchitecture, 225-236, Dec. 2001.

[26] R. Heald et al., “A Third-Generation SPARC V9 64-b Microprocessor”, IEEE Journal of Solid-State Circuits, 35(11): 1526-1538, Nov. 2000.

[27] J. L. Hennessy, D. Goldberg, K. Asanovic and D. A. Patterson, Computer Architecture: Quantitative Approach, 3rd Edition, Morgan Kaufmann Publication, 2003.

[28] A. Henstrom, “Scheduling Operations Using a Dependency Matrix”, United States Patent 6557095, Apr. 2003.

[29] P. Hicks, M. Walnock and R. Owens, “Analysis of Power Consumption in Memory Hierarchies”, International Symposium on Low-Power Electronics Design, 239-244, Aug. 1997.

[30] D. A. Hodges, H. G. Jackson, and R. A. Saleh, Analysis and Design of Digital Integrated Circuits, 3rd ed., McGraw-Hill, 2004.

[31] E. Jacobsen, E. Rotenberg and J. E. Smith, “Assigning Confidence to Conditional Branch Predictions”, International Symposium on Microarchitecture, 142-152, Dec. 1996.

[32] M. Kamble and K. Ghose, “Analytical Energy Dissipation Models for Low Power Caches”, IEEE International Symposium on Low-Power Electronics Design, 143-148, Aug. 1997

[33] R.E. Kessler, “The Alpha 21264 Microprocessor”, IEEE MICRO, 19(2):24-36, Mar. 1999.

[34] G. Kucuk, O. Ergin, D. Ponomarev and K. Ghose, ”Reducing Power Dissipation of Register Alias Tables in High-Performance Processors”, IEE Proceedings on Computers and Digital Techniques, 152(6): 739-746 , Nov. 2005.

[35] Y. Li, and J. Henkel, “A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems”, Design Automation Conference, 188-193, Jun. 1998.

[36] X. Liang and D. Brooks, “Highly Accurate Power Modeling Method for SRAM Structures with Simple Circuit Simulation”, The Second Watson Conference on Interaction between Architecture, Circuits, and Compilers (p=ac2), Sept. 2005.

[37] X. Liang , K. Turgay and D. Brooks, “Architectural Power Models for SRAM and CAM Structures Based on Hybrid Analytical/Empirical Techniques”, IEEE/ACM International Conference on Computer-Aided Design, 824-830, Nov. 2007

References

[38] M. Mamidipaka, K. Khouri and N. Dutt, "A Methodology for Accurate Modeling of Energy Dissipation in Array Structures", IEEE International Conference on VLSI Design, 320-325, Jan. 2003.

[39] M. Mamidipaka, K. Khouri, N. Dutt and M. Abadir, "IDAP: A Tool for High-Level Power Estimation of Custom Array Structures", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(9):1361-1369, Sept. 2004.

[40] M. Mamidipaka and N. Dutt, "eCACTI: An Enhanced Power Estimation Model for On-chip Caches", Technical Report, Center for Embedded Computer Systems, University of California at Irvine, Oct.2004.

[41] M. Mamidipaka, K. Khouri, N. Dutt and M. Abadir, "Analytical Models for Leakage Power Estimation of Memory Array Structures", IEEE/ACM/IFIP International Conference on Hardware/Software Co-design and System Synthesis, 146-151, Sep. 2004.

[42] M. Margala, "Low-power SRAM circuit design", IEEE Workshop on Memory Technology, Design and Testing, 115-122, Aug. 1999.

[43] A. Moshovos, "Power-Aware Register Renaming", Technical Report, Electrical and Computer Engineering Department, University of Toronto, Aug. 2002.

[44] A. Moshovos, "RegionScout: Exploiting Coarse-Grain Sharing in Snoop-Coherence", International Symposium on Computer Architecture, 234-245, Jun. 2005.

[45] A. Moshovos, G. Memik, B. Falsafi and A. Choudhary, "Jetty: Filtering Snoops for Reduced Energy Consumption in SMP Servers", International Conference on High-Performance Computer Architecture, 85-96, Feb. 2001.

[46] A. Moshovos, "Checkpointing Alternatives for High Performance, Power-Aware Processors", IEEE International Symposium on Low Power Electronics and Design, 318-321, Aug. 2003.

[47] J. M. Mulder, N. T. Quach and M. J. Flynn, "An Area Model for On-chip Memories and its Application", IEEE Journal of Solid-State Circuits, 26:98-106, Feb. 1991.

[48] H. E. Neil Weste and D. Harris, Principles of CMOS VLSI Design, 3rd ed., Addison Wesley, 2004.

[49] S. Palacharla, "Complexity-Effective Superscalar Processors", Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1998.

References

[50] S. Paracharla, N. P. Jouppi and J. E. Smith, "Quantifying the Complexity of Superscalar Processors", Technical Report, University of Wisconsin-Madison, Nov. 1996.

[51] S. Paracharla, N. P. Jouppi, and J. E. Smith, "Complexity Effective Superscalar Processors", IEEE/ACM International Symposium on Computer Architecture, 25(2): 206-218, May 1997.

[52] D. Parikh, K. Skadron, Y. Zhang, M. Barcella and M.R. Stan, "Power Issues Related to Branch Prediction", International Symposium on High-Performance Computer Architecture, 233-244, Feb. 2002.

[53] D. Parikh, K. Skadron, Y.Zhang and M. Stan, "Power-aware Branch Prediction: Characterization and Design", IEEE Transactions on Computers, 53(2):168-186, Feb. 2004.

[54] J. K. Peir, S.C. Lai, S.L. Lu, J. Stark and K. Lai, "Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching", International Conference on Supercomputing , 189-198, Jun. 2002.

[55] D. Ponomarev, G. Kucuk and K. Ghose, "AccuPower: an Accurate Power Estimation Tool for Superscalar Microprocessors", Design, Automation and Test in Europe Conference and Exhibition, 124-129, Mar. 2002.

[56] G. Reinman and N. Jouppi, "CACTI 2.0: An Integrated Cache Timing and Power Model", WRL Research Report 2000/7, Feb. 2000.

[57] R. Sangireddy, "Reducing Rename Logic Complexity for High-Speed and Low-Power Front-End Architectures", IEEE Transactions on Computers, 55(6):672- 685, Jun. 2006.

[58] P.G. Sassone, J. Rupley II, E. Brekelbaum, G. H. Loh and B. Black ,"Matrix Scheduler Reloaded" , International Symposium on Computer Architecture, 335-346 , May 2007

[59] E. Schmidt, G. von Colln, L. Kruse, F. Theeuwen, and W. Nebel, "Memory Power Models for Multilevel Power Estimation and Optimization", IEEE TVLSI, 10(2):106-108, 2002.

[60] S. Sethumadhavan, R. Desikan, D. Burger, C.R. Moore and S.W. Keckler, "Scalable Hardware Memory Disambiguation for High-ILP Processors", IEEE MICRO, 24(6):118-127, Nov. 2004.

[61] J. Shen and M. Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors, McGraw-Hill Publication, 2005.

[62] P. Shivakumar and N. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model", WRL Research Report 2001/2, Aug. 2001.

References

[63] M. R Stan, “Synchronous Up/Down Counter with Clock Period Independent of Counter Size”, IEEE Symposium on Computer Arithmetic, 274-281, Jul. 1997.

[64] M.R. Stan , A. F. Tenca and M. D. Ercegovac, “Long and Fast Up/Down Counters”, IEEE Transactions on Computers, 47(7):722-735, Jul. 1998.

[65] D. Tarjan, S. Thoziyoor and N. P. Jouppi, “CACTI 4.0”, HP Labs Technical Report, HPL-2006-86, 2006.

[66] S. Thoziyoor, N. Muralimanohar and N. P. Jouppi, ”CACTI 5.0”, Advanced Architecture Laboratory HP Laboratories HPL-2007-167, Oct. 2007

[67] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim and W. Ye, “Energy Driven Integrated Hardware-Software Optimizations Using Simplepower”, International Symposium on Computer Architecture, 95-106, Jun. 2000.

[68] T. Wada, S. Rajan and S. A. Przybylski, “An Analytical Access Time Model for On-Chip Cache Memories”, IEEE Journal of Solid-State Circuits, 27:1147–1156, Aug. 1992.

[69] S. Wilton and N. Jouppi, “An Enhanced Access and Cycle Time Model for On-Chip Caches”, WRL Technical Report 93/5, June 1994.

[70] K. C. Yeager, “The MIPS R10000 Superscalar Microprocessor”, IEEE MICRO, 16(2):28-40, Apr. 1996.

[71] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron and M. Stan. “Hotleakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects”, Technical Report CS-2003-05, University of Virginia, Mar. 2003.

[72] V. Zyuban and P. Kogge, “The Energy Complexity of Register Files”, International Symposium on Low-Power Electronics Design, 305-310, Aug. 1998.

[73] V.V. Zyuban and P. M. Kogge, “Inherently Lower-Power High-Performance Superscalar Architectures”, IEEE Transactions on Computers, 50(3): 268-285, Mar. 2001.

[74] V. Zyuban, “Inherently Lower-Power High Performance Superscalar Architectures”, PhD Thesis, Department of Computer Science and Engineering, University of Notre Dame, 2000.