TECHNIQUES TO ADDRESS UNRELIABILITY AND VARIABILITY OF COMPUTING SYSTEMS

BY

JUN NAKANO

B.S., University of Tokyo, 1988
M.S., University of Tokyo, 1990

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

As technology feature size continues to shrink, we see two challenging problems in the design of computer systems. One is the hardware unreliability due to increasing chances of transient hardware faults caused by high-energy particles and temperature hot spots. The other is the variability in the semiconductor manufacturing process, which manifests itself as a large variation in gate lengths, threshold voltage, and other parameters within a wafer and even within a die. This variability eventually impacts the frequency and the leakage power dissipation of a chip.

In the first part of this thesis, we study the problem of handling I/O in memory-based checkpointing systems. The increasing demand for reliable computers has led to proposals for hardware-assisted rollback of memory state. Such approach promises major reductions in Mean Time To Repair (MTTR). The benefits are especially compelling for database servers, where existing recovery software typically leads to downtimes of tens of minutes. Unfortunately, adoption of such proposals is hindered by the lack of efficient mechanisms for I/O recovery.

We present and evaluate ReViveI/O, a scheme for I/O undo and redo that is compatible with mechanisms for hardware-assisted rollback of memory state. We have fully implemented a Linux-based prototype that shows that low-overhead, low-MTTR recovery of I/O is feasible. For 20–120 ms between checkpoints, a throughput-oriented workload such as TPC-C has negligible overhead. Moreover, for 50 ms or less between checkpoints, the response time of a latency-bound workload such as WebStone remains tolerable. In all cases, the recovery time of ReViveI/O is practically negligible. The result is a cost-effective highly-available server.

In the second part of this thesis, we study architecture-aware fine-grain adaptive body biasing to improve the frequency and leakage power dissipation of processor chips. As VLSI technology continues to scale, parameter variation is about to pose a major challenge to high-performance

processor design. In particular, the within-die variation of threshold voltage is directly detrimental to the chip's frequency and leakage power. One proposed technique to address such variation is Fine-Grain Body Biasing (FGBB), where different chip sections called cells are given a certain voltage bias that modifies the threshold voltage.

The insight of this work is that FGBB should be applied in an *architecture-aware* manner — using cells that follow the shapes of architectural modules such as execution units or caches. The reason is that architectural functionality affects the BB needed through *temperature* and *type of critical path*. To prove this idea, we develop a novel model of threshold voltage variation and apply it to simulated batches of chips. We show that architecture-aware FGBB (AA-FGBB) enables 35% of the chips to work at the highest frequency, compared to 18% with advanced conventional FGBB — potentially increasing each chip's value by 50%. It also reduces the leakage of the chips by 40%, compared to 25% with advanced conventional FGBB. Overall, computer architecture plays an essential role in mitigating parameter variations.

To my parents Takeshi and Akiko Nakano, and my mentor, Toyohiko Ohga.

Acknowledgments

I would like to thank my advisor, Professor Josep Torrellas for his continuous encouragement and advice on my research. Without his help, I cannot imagine how I could have come this far to complete my study. I would also like to thank Professors Marc Snir, Saburo Muroga, Yuanyuan Zhou and Dr. Kourosh Gharachorloo for being in my committee and for their helpful suggestions.

For the ReViveI/O project, I am very grateful to Dr. Gharachorloo for providing me the opportunity to work in the Western Research Laboratory. He is a great mentor and motivated me to pursue this project from the viewpoint of practical application to the commercial products. He also gave me valuable advice in many occasions during the project. Pablo Montesinos helped me complete the experiments with his expertise in Linux kernel programming. I also thank Milos Prvulovic (now a professor at Georgia Tech) who first came up with the idea of hardware-based frequent memory-checkpointing, which is the source of this project.

In the variability project, I benefited a lot from discussion and help from Radu Teodorescu and Abhishek Tiwari. In this new and fast-moving research area, I could not have attained good results in a timely manner without their support. It is interesting to notice that the importance of this problem was already hinted in the lectures that I took from Professor Muroga a few years ago but it took me a while to realize it. He is also my mentor and he and his wife Yoko helped me start a new life in Champaign-Urbana back in 2000.

I also thank the rest of the extended I-ACOMA group: Wonsun Ahn, Sujoy Basu, Qiang Cao, Luis Ceze, Marcelo Cintra, Sheila Clark, Maria Garzaran, Brian Greskamp, Michael Huang, Wei Liu, José Martínez, Jose Renau, Paul Sack, Smruti Sarangi, Yan Solihin, Karin Strauss, and James Tuck. I enjoyed my graduate life very much thanks to excellent research environment of Department of Computer Science, at Digital Computer Laboratory (2000-2004) and at Siebel

Center for Computer Science (2004-2005). Finally, I would like to thank IBM Japan for giving me the opportunity to pursue a Ph.D. degree, especially my ex-boss and mentor, late Mr. Toyohiko Ohga, who encouraged and supported me to do so.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DD	Device Driver.
PDD	Pseudo Device Driver.
MR fault	Memory-Recoverable fault.
NMR fault	Non-Memory Recoverable fault.
BB	Body Bias.
FGBB	Fine-Grain Body Bias.
AA-FGBB	Architecture-Aware Fine-Grain Body Bias.
FBB	Forward Body Bias.
RBB	Reverse Body Bias.

List of Symbols

$\mathcal{F}$	Target frequency (for body biasing).
ϕ	Range (in the spherical model).
I_{leak}	Leakage current.
k	Boltzman constant (1.38×10^{-23} J/K).
L	Gate length
μ	Average threshold voltage.
q	Electron charge (1.60×10^{-19} C).
σ	Standard deviation of the threshold voltage.
T	Temperature.
T_{cp}	Critical path delay.
T_g	Gate delay.
V	Supply voltage.
V_{th}	Threshold voltage.

Chapter 1

Introduction

Technology scaling poses serious challenges to computer designers and manufacturers. The most notable problems are those related to reliability and variability.

It is believed that the rate of single-event upsets (SEU) — bit flip errors due to hits by cosmic rays or radiation from the packaging material — per bit will stay more or less constant. This is because the effects of the shrinking cross section and the decrease in Q_{crit} (minimum charge needed to flip the state of a memory structure) cancel each other despite technology scaling. Combined with Moore's law, this implies that the rate of SEUs per system will grow exponentially. In addition, it is also believed that temperature hot spots may cause bit flips in future technologies. The problem of reliability can only be solved by means of redundancy (in time, space, or both). Among the proposed reliability mechanisms, ReVive [61] is particularly interesting in that it has low redundancy (N+1 in space and 6% in time) and high availability (five 9's). It has been proposed for a distributed shared-memory multiprocessor system that frequently checkpoints the system state in memory. However, since ReVive does not support I/O, the mechanism is not readily applicable to commercial systems. The main obstacle is *the output commit problem*. What if a machine sends a message to a client and then rolls back to a state prior to sending the message? It is against the causality and can also lead to an incorrect result if the re-execution is not deterministic. In this thesis, we present *ReViveI/O*, a mechanism that adds supports for I/O to ReVive, thus completing the full-system support for ReVive (and similar hardware-based memory-checkpointing systems). We focus on the disk and network I/O because they are crucial for back-end database servers, which interact with the outside world mostly through disk and network I/O. Chapter 2 elaborates on how I/O can be efficiently supported in hardware-based memory-checkpointing systems. The proposed scheme, ReViveI/O, mainly targets throughput-oriented workloads. It provides a fast I/O recovery

mechanism that can be applied to a general class of hardware-based memory-checkpointing systems and incurs very small fault-free execution overhead. ReViveI/O was implemented and tested on multiprocessor Linux servers.

Variability is another challenge caused by technology scaling. Since the 180 nm technology node, the feature size has always been smaller than the wavelength that is used in the lithography. This results in inaccuracy in the printed circuits. In addition, as the number of dopant atoms (N) in a channel decreases with the technology scaling, the random fluctuations, which are $O(\sqrt{N})$, become no longer negligible. These *process variations* lead to variation in the threshold voltage of MOSFETS, which in turn reduces the chip frequency and increases the subthreshold leakage power dissipation. Although we can neither fix the printed circuit patterns nor change the number of implanted dopant atoms, we can effectively control the resulting threshold voltage variation by a technique called *body biasing*. Chapter 3 describes body biasing and shows efficient ways to use body biasing to cope with process (P) and temperature (T) variation. Our work is one of the first to study body biasing with both P and T variations taken into consideration. Our results clarify the importance of architecture-awareness when applying body bias in a fine-grain manner.

Chapter 2

ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers

2.1 Introduction to ReViveI/O

Highly-available shared-memory servers have to be able to cope with system-level faults. Faults are often transient, such as hardware glitches caused by high-energy particles, or OS panic due to unusual interleavings of software events. There are also permanent hardware faults, which can bring down part of the machine. Fault frequencies are projected to remain high in the future. This is worrisome, given the growing number of businesses with database applications that crucially depend on their servers being up practically all the time.

One approach to attain fault tolerance is to employ extensive self-checking and correcting hardware, often through redundancy and even lock-step execution. This is the approach used by HP's Nonstop Architecture [25] and IBM's S/390 mainframes [71]. Unfortunately, this approach is too expensive for many users.

An alternative approach is to use plain server hardware and support software-based checkpoint and rollback recovery. In such systems, the operating system [41, 44, 66], virtual machine monitor [9], or application (e.g., the database [21]) periodically checkpoints the state of the machine, virtual machine, or processes, respectively, to safe storage. If a fault is detected, the system rolls back to a state preceding the fault. However, since software checkpointing has significant overhead, checkpoints are typically only taken every few minutes or less frequently. As a result, when a fault occurs, the Mean Time To Repair (MTTR) is significant, and the machine becomes unavailable for a sizable period. For example, the recovery time of Oracle 9.2 on a Solaris server is typically tens of minutes [49].

A second shortcoming of software-based checkpointing appears in workloads where server and

clients frequently exchange messages. To correctly support recovery, the server must delay sending messages until after they are checkpointed. If checkpoints are infrequent to minimize overheads, messages suffer long delays.

One way to significantly reduce server MTTR and avoid long message delays is to support *high-frequency* checkpointing (e.g., one every few tens of milliseconds). Several architectures with such support have been proposed [48, 52, 61, 72]. These architectures rely on hardware assistance for checkpointing or for data buffering, logging or replication. For example, ReVive induces about 6% overhead and recovers from the types of faults supported in less than 1 second [61]. Such tiny MTTR boosts machine availability. Moreover, as suggested by the ROC project, it opens up opportunities to lower cost of ownership [59].

Unfortunately, past work on these high-frequency checkpointing architectures has focused on recovering the *memory state* of the machine. It has not fully addressed the problem of rollback recovery in the presence of I/O. When workloads perform I/O, rollback is tricky: how can the server "undo" a disk write or a message send? Can it "redo" it? Unless these issues are addressed, the proposed high-frequency checkpointing solutions are unusable. These issues are also particularly relevant to architectures for transactional memory [23], which rely on the ability to roll back a section of code and then re-execute it.

A known approach to handle I/O in checkpointing systems is to delay the commit of output until the next checkpoint (output commit problem). To accomplish this, Masubuchi *et al.* [48] proposed adding a "virtual" or "Pseudo" Device Driver (PDD) layer between the kernel and the Device Drivers (DD). Disk output requests are redirected to the PDD rather than the DD. The PDD blocks any output-requesting process until the next checkpoint [47], after which the output is performed. Masubuchi *et al.*'s design has limitations, such as (i) blocking processes until a checkpoint and (ii) only supporting disk I/O. However, their general method is attractive because it requires no kernel or application modification. It can be built upon to provide efficient I/O undo/redo for high-frequency checkpointing architectures.

2.1.1 Contribution of This Work

Our main contribution is the full implementation, testing, and experimental evaluation of *ReViveI/O*, an efficient I/O undo/redo scheme that is compatible with high-frequency checkpointing architectures such as ReVive [61] and SafetyNet [72]. Our work completes the viability assessment of such novel memory-recovery architectures. It is only through a complete implementation that we identify true overheads, relevant ordering constraints, and corner cases. Moreover, we perform a sensitivity analysis of what checkpoint frequencies are required to maintain acceptable throughput and tolerable response times.

We also enhance Masubuchi *et al.*'s approach in two ways. First, the PDD now also supports network I/O. Secondly, the disk PDD, rather than blocking the output-requesting process, quickly buffers the output and returns. After the next checkpoint, the I/O operation is committed in the background. This provides efficient I/O undo/redo.

We installed our ReViveI/O prototype on a Linux 2.4-based multiprocessor server running TPC-C on Oracle, and WebStone on Apache. Our prototype shows that low-overhead, tiny-MTTR recovery of I/O is feasible. Specifically, for 20–120 ms between checkpoints, a throughput-oriented workload such as TPC-C has negligible overhead. In addition, for 50 ms or less between checkpoints, the response time of a latency-bound workload such as WebStone on Apache remains tolerable. In all cases, the recovery time of ReViveI/O is practically negligible. Finally, combining ReVive and ReViveI/O is likely to reduce the throughput of TPC-C-class applications by 7% or less for 60–120 ms checkpoint intervals, while incurring a tiny MTTR of less than 1 second.

Our work is significant in that, with ReVive and ReViveI/O, a shared-memory server can quickly recover from: (i) any hardware (and some software) transient faults in the machine, and (ii) permanent faults that at most take out one node in the machine. Indeed, both the processor/memory state (thanks to ReVive) and the I/O state (thanks to ReViveI/O) are restored to the preceding checkpoint *within 1 second and transparently* to the database. No ongoing database transactions are lost.

There are rare faults for which ReVive cannot restore the processor/memory state, such as the simultaneous permanent loss of multiple nodes. In this case, the fault is not transparent to

the database. A few seconds after the machine is rebooted, ReViveI/O brings the I/O state to its correct state at the preceding checkpoint. Then, we simply depend on the normal recovery mechanisms of the database to reconstruct the state from the logs saved on disk.

The overall result is much higher server availability: the majority of faults are recovered from with sub-second MTTR and transparently, while only infrequent faults require the much slower recovery mechanism of the database.

2.2 Background

2.2.1 Context of Our Work

The context of our work is shared-memory multiprocessors such as IBM's eServer pSeries p5 595 [29] or HP's Integrity Superdome [26] used as back-end database servers. These servers store the database in local disk subsystems and communicate over networks with many clients. They execute transaction-processing applications similar to TPC-C.

A major issue in these systems is server uptime. Unfortunately, a high-energy particle impact may cause a processor reset, an unusual data race may crash the OS, or a link failure may disconnect a node. In these cases, transactions are typically aborted and the database attempts to recover. Such recovery often renders the server unavailable for tens of minutes [49].

To understand the recovery requirements of these systems, note that I/O is practically limited to disk and network. Moreover, these workloads are typically not latency bound. For example, in TPC-C, 88% of transactions are NewOrder or Payment, which involve the exchange of a single request and response between client and server. IBM's p5 595 reports an average response time of 340 ms for these transactions [79]. Consequently, adding a few tens of ms to each transaction to support a recovery scheme is tolerable.

2.2.2 Fault Model

We leverage proposed rollback-recovery architectures [48, 52, 61, 72] that support high-frequency checkpointing (ten times or more per second) and, for the fault types supported, are able to recover the *memory state* of the machine before the fault. These schemes typically have low overhead and

a tiny MTTR.

As an example, we use ReVive [61] in this study. Appendix A outlines ReVive. With 100 ms between checkpoints, ReVive has an average execution overhead of 6.3%. Moreover, it recovers from the supported faults in under 1 second. This results in 99.999% availability even with one fault per day.

Specifically, ReVive recovers the memory state of the machine for: (i) transient faults and (ii) permanent faults that at most take out one node in the machine. Although fault detection is beyond the scope of this study, the implicit assumption is that some mechanism detects these faults within a checkpoint interval. Such short detection latency is more feasible for hardware faults than for software ones. However, there are some software transient faults that are fail fast. For example, Gu *et al.* [22] show that a sizable portion of kernel errors can be detected within 100,000 cycles. Overall, we refer to all these faults, from which ReVive can recover the memory state, as *Memory-Recoverable* (MR) faults.

The other faults, from which ReVive cannot recover the machine's memory state, we call *Non-Memory-Recoverable* (NMR). An example is the simultaneous permanent loss of multiple nodes [61].

In this study, we also assume that non-volatile storage, namely disks and any closely-attached non-volatile memories (NVRAMs), can only suffer transient faults. They have the appropriate support (e.g., RAID 5) to avoid permanent faults.

With these assumptions, we will show that, for MR faults, we restore both the processor/memory state (thanks to ReVive) and the I/O state (thanks to ReViveI/O) to the preceding checkpoint. The restoration is *transparent* to the database. No ongoing transactions are lost (Figure 2.1).

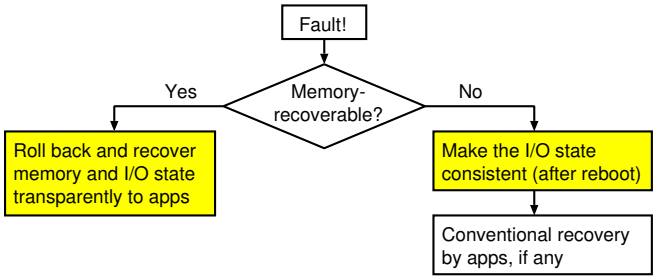

Figure 2.1: Faults handled by the combination of ReVive and ReViveI/O.

For NMR faults, recovery is not transparent to the database. The machine has to be fixed and rebooted. ReViveI/O then restores the I/O state to its consistent state at the preceding checkpoint. Finally, the conventional recovery mechanisms of the database reconstruct the database state.

2.2.3 Integrating I/O with Checkpoints

Work on checkpointed message-based distributed systems [17, 35] shows how to support I/O undo/redo under checkpointing. The commit of outputs is delayed until the next checkpoint (output commit problem); only then can the system guarantee that it will not have to roll back to a state prior to issuing the outputs.

To address the output commit problem without kernel modifications, Masubuchi *et al.* proposed the Pseudo Device Driver (PDD) [48] (Figure 2.2). Disk output requests are redirected to the PDD rather than the Device Driver (DD). The PDD blocks any output-requesting process until the next checkpoint [47], after which the output is performed. The PDD can be considered an extremely thin virtual machine layer for I/O checkpointing.

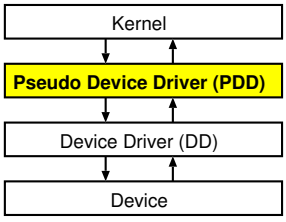

Figure 2.2: The Pseudo Device Driver software layer.

We enhance Masubuchi *et al.*'s scheme in two ways. First, processes requesting disk writes are not blocked until the next checkpoint. Secondly, we also support network I/O. Kernel, DDs, and server/client applications remain unmodified.

2.3 Architecture of the Prototype

This section describes the organization ReViveI/O, with key ordering issues, overheads, and limitations.

2.3.1 Description of Operation

We start by examining three properties that we leverage. Then, for readability, we describe ReViveI/O in two steps: first, an initial incomplete solution, and then the complete one.

Properties Leveraged

We leverage three properties to build a low-overhead I/O undo/redo prototype. First, ReVive's ability to roll back the memory state is leveraged to restore PDD consistency after a fault. Specifically, we assign to the PDD a portion of main memory called the *Memory Buffer*. In there, the PDD buffers all the output requests until the next checkpoint; after the checkpoint, the outputs are performed in the background and removed from the buffer. If a fault occurs, ReVive returns the memory state to the previous checkpoint. This automatically makes the PDD consistent: all the output requests in the current checkpoint interval disappear from the Memory Buffer, and all those from the previous checkpoint interval re-appear in the Memory Buffer and are ready to be performed again.

Second, the fact that the output operations under consideration are idempotent (i.e., replayable) is leveraged to allow the recovery scheme to re-perform output operations without hurting correctness. Indeed, disk output operations are trivially idempotent. Network output is idempotent due to the high-level support provided by TCP [77]. With TCP, each packet has a sequence number. If the client receives the same packet twice, TCP sees the same sequence number and discards one of them[1]. Consequently, correctness is not compromised when, after a rollback, the requests in the Memory Buffer force our scheme to re-write the same disk blocks and re-send the same messages.

Finally, properties of the I/O considered are leveraged to not have to buffer any inputs for later "re-consumption" should rollback be needed. Specifically, disk inputs need no buffering because the application will automatically re-issue them if it needs to. For network input, we avoid buffering by again relying on TCP properties. With TCP, packets are acknowledged by the receiver; if the client does not receive an acknowledgment (ACK) from the server within a timeout period, it resends the

[1]The User Datagram Protocol (UDP) does not provide TCP's support to eliminate duplicates. UDP is unreliable by definition, and the application (e.g., NFS over UDP) is responsible for dealing with duplicate and lost packets. Consequently, we only focus on TCP.

packet. In our design, ACKs, like all outgoing messages, are delayed by the server until after the next checkpoint.

Consequently, suppose that the server receives an input message, issues an ACK that gets stored in the Memory Buffer, and a fault occurs. Two cases are possible. First, if the fault occurs before the end of the next checkpoint, the rollback removes the effect of the input message from the server, as well as the ACK from the Memory Buffer. In this case, the ACK is not sent and the client will resend the input message. If, instead, the fault occurs after the next checkpoint, the rollback removes neither the effect of the input message from the server, nor the ACK from the Memory Buffer. This case is also correct because the ACK will eventually be sent. In either case, the server does not need to buffer network input.

Initial Incomplete Solution: BufferVolatile

All network and disk output requests issued by the application (*OutReq1* and *OutReq2* in Figure 2.3-(a)) are transparently intercepted by the PDD and buffered in the Memory Buffer. The buffered information includes the output data and metadata such as the destination block number in the device. After the next checkpoint (*C2* in Figure 2.3-(a)), the PDD passes the information to the DDs, which perform the output operations (e.g., DMA writes to disk or to the network card) in the background (*OutOp1* and *OutOp2* in Figure 2.3-(a)).

Consider now input requests, such as reads from the disk or the network card. On receiving the request (*InReq1* in Figure 2.3-(d)), the PDD checks if the requested data is in the Memory Buffer. If so, the data is provided. Otherwise, the PDD passes the request to the DD, which performs the operation in the background (*InOp1* in Figure 2.3-(d)). As indicated in Section 2.3.1, no buffering is needed.

We call this initial solution *BufferVolatile*. With it, if an MR fault (Section 2.2.2) occurs, the server recovers both memory and I/O states *transparently* to the running application. Consider the four possible timeframes wherein a fault can occur.

1. Fault before the end of the checkpoint that immediately follows the I/O request (*Fault 1* in Figures 2.3-(b) and (e)). In this case, ReVive rolls back the memory state to the previous checkpoint *C1*. As a result, the Memory Buffer loses any record of output request *OutReq1*. This

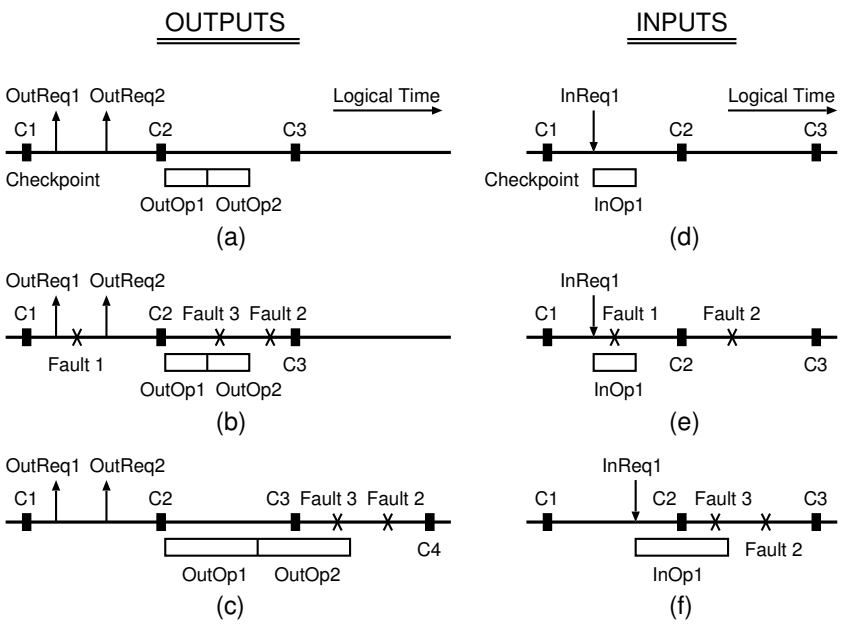

Figure 2.3: I/O operations and faults in different scenarios. In the figure, *InReq*, *OutReq*, *InOp*, and *OutOp* mean input request, output request, input operation, and output operation, respectively.

automatically "undoes" *OutReq1*, as desired. Thus, *OutOp1* is not performed. As for input I/O, since the rollback operation involves resetting the devices, any ongoing input operation such as *InOp1* is aborted.

2. Fault after the end of the checkpoint that immediately follows the I/O request; the I/O is already performed (*Fault 2* in Figures 2.3-(b) and (e)). ReVive rolls back the memory state to the previous checkpoint *C2*. The only interesting case is for outputs. The Memory Buffer gets restored to the state it had at *C2*, where it contained a record of the output operations to perform. Consequently, the PDD will eventually automatically re-issue *OutOp1* and *OutOp2* to the DDs. This is correct because of the idempotent nature of the I/O in consideration.

3. Like Case 2 but the background I/O is not yet completely performed when the fault occurs (*Fault 3* in Figure 2.3-(b)). As the system rolls back, the devices are reset and the ongoing I/O is aborted. Then, all I/O operations (*OutOp1* and *OutOp2*) will eventually be performed again.

4. Special case: Fault in an interval preceded by a checkpoint overlapped with an I/O operation. Sometimes, an I/O operation initiated before a checkpoint extends past it. This is

seen for *OutOp2* in Figure 2.3-(c) and *InOp1* in Figure 2.3-(f). If a fault such as *Fault 2* or *Fault 3* in these figures occurs, the memory state rolls back to the checkpoint that overlapped with the I/O operation. During the recovery process, the I/O operation (*OutOp2* or *InOp1*) gets killed, since all I/O devices (disk controller and network adapter) get reset. This is discussed in Section 2.3.3. Unfortunately, the rollback would leave the memory state in inconsistent state: while the I/O operation is killed, it is incorrectly marked "in progress", and it is only partially performed.

To solve this problem, after the recovery process rolls back the memory state to the previous checkpoint, the PDD re-issues to the DDs any checkpoint-overlapping I/O operation *from the beginning*. Note that the PDD can find out what are the I/O operations that are (incorrectly) marked "in progress" (*OutOp2* and *InOp1*). Optionally, the PDD can skip re-issuing the network input operations that overlapped with the checkpoint: there is no need to re-initiate the transfer of data from the network adapter to memory because TCP will ensure that incoming packets are retransmitted.

Appendix A gives more formal argument about *BufferVolatile* using state transition diagrams. Overall, our *BufferVolatile* scheme ensures database consistency in an environment with MR faults (and/or transient faults in non-volatile storage as per Section 2.2.2). For example, assume that a client starts a transaction that involves writes to disk (Figure 2.4-(a)). After the disk PDD has buffered the data and destination block number in the Memory Buffer, the database sends a response message to the client. The message is buffered by the network PDD. After the next checkpoint, the message is sent and the write is issued to the disk. If an MR fault occurs before this checkpoint, *BufferVolatile* will roll back and eventually receive the automatic retransmission of the original request message after the timeout. Moreover, if the PDD observes a transient disk error when it issues the write, it will simply retry until it succeeds. In any case, when the client receives the response message, it can assume that the disk write in the transaction has been committed. Some ordering issues are examined in Section 2.3.2.

Complete Solutions: Buffer and Rename

BufferVolatile is inadequate if an NMR fault (Section 2.2.2) occurs, such as the simultaneous permanent loss of multiple nodes. As an example, consider Figure 2.4-(a) after the second checkpoint.

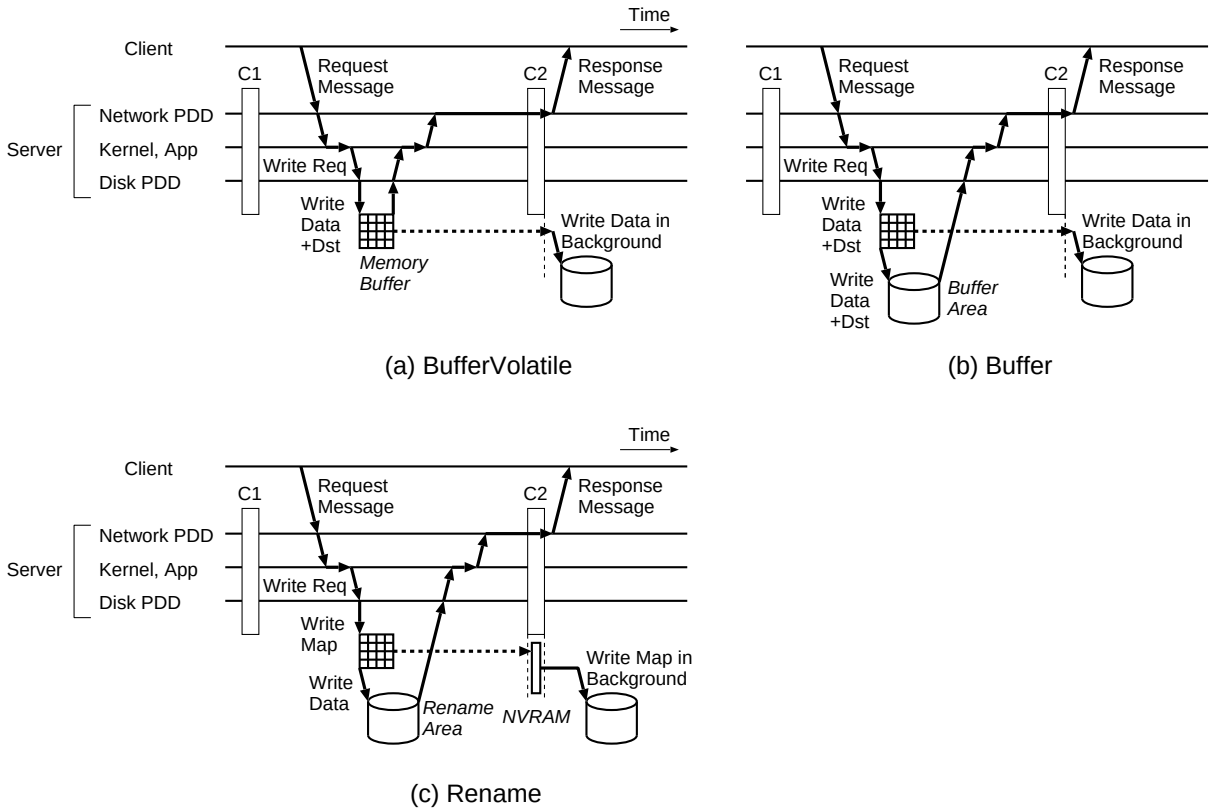

Figure 2.4: Operation of ReViveI/O. The scenario depicts a transaction with a disk write. NVRAM and *Dst* stand for Non-Volatile RAM and the destination block number, respectively.

Suppose that an NMR fault occurs after the response to the client is sent but before the disk is updated in the background. ReVive cannot restore the contents of the Memory Buffer and, therefore, re-issue the buffered write to disk. The disk is left in a state that is inconsistent with the information passed to the client. Note that conventional database recovery mechanisms cannot help: the missing write can be a log write, without which the database cannot redo the operation.

To solve this problem, we enhance *BufferVolatile* to ensure that the PDD also saves the output request information in non-volatile "temporary" storage before the next checkpoint. If an NMR fault occurs as in the example just described, we can copy the information from the non-volatile temporary storage to the server disk, and thus make the disk consistent. Then, we can rely on the conventional mechanisms of the database for recovery, although the downtime will be longer (Figure 2.1).

We propose two alternative schemes, called *Buffer* and *Rename*, as shown in Figures 2.4-(b) and (c), respectively. *Buffer* is based on temporarily buffering output request data, and is conceptually simpler. *Rename* is based on renaming the data, and can be more efficient because it requires fewer disk writes. To describe the schemes, we focus on disk I/O because network I/O does not distinguish between the schemes.

In *Buffer*, a disk write request updates the Memory Buffer and a disk buffer area before returning (Figure 2.4-(b)). The update includes both the data and some metadata such as the destination block number. To speed up this operation, the updates to the disk buffer are done on sequential blocks. Moreover, the disk buffer can be a dedicated small, fast disk, similar to the Disk Caching Disk [28]. After the next checkpoint, all data in the Memory Buffer are copied to their true locations on the main disk. In fault-free conditions, the disk buffer is never read.

In *Rename*, a write request writes the output data to a new disk block in a rename area, and saves the new logical-to-physical block number mapping in the Memory Buffer before returning (Figure 2.4-(c)). During the checkpoint, the mappings in the Memory Buffer are copied to a small (e.g., 32 MB) Non-Volatile RAM (NVRAM) associated with the disk. Later, the mappings in the NVRAM are committed to disk in the background.

The NVRAM is not a single point of failure. Specifically, there are commodity disk adapters with internal NVRAM where the NVRAM is transparently backed up by an additional copy of the

data. An example is IBM's Fast Write Cache [30]. Moreover, transient errors in the NVRAM or associated disk are handled by the PDD retrying the request. Recall that we assume there are no permanent faults in non-volatile storage.

Note that these NVRAMs usually work asynchronously — the data is destaged to the disk only when the NVRAM is getting full or the data has stayed in the NVRAM for a certain time. Therefore, if a fault occurs during a checkpoint, an asynchronous NVRAM could incorrectly write its mappings to disk while we are rolling back to the previous checkpoint. This problem is solved by also storing the original mapping information in the NVRAM, so that if the problem occurs, we can undo the changes of mapping.

An important design issue in *Rename* is the policy for allocating the renamed blocks. One option is to write the new blocks sequentially on a free disk area, just like the log-structured file system (LFS) [65]. With this design, occasional defragmentation of the disk may be needed. Another option is to map each block to either one of two physically consecutive blocks in the disk as in TWIST [63]. This design requires double disk space for blocks. However, the mapping information per block is only one bit, compared to (typically) 8 bytes for the first design.

With *Buffer* or *Rename*, the disk can always be brought to the state corresponding to the checkpoint immediately preceding the fault, even for an NMR fault. Indeed, consider a disk write request. If the fault happens between the request and the end of the next checkpoint, the contents of the disk buffer are discarded (*Buffer*) or the new mapping information is not written to NVRAM (*Rename*); the main disk remains unmodified and the client is never notified. If, instead, the fault happens after the next checkpoint, a disk update is guaranteed to occur: the data and metadata in the disk buffer (*Buffer*) or the mapping in NVRAM (*Rename*) are used to update the disk.

2.3.2 Ordering Issues

Our schemes (*Buffer* and *Rename*) change the timing of I/O operations. However, they always satisfy the ordering properties of I/O in databases, which require that logs are fully committed before data is. This is accomplished by updating the Memory Buffer and the disk buffer or rename area in the order that the disk write requests are received. Moreover, after the subsequent checkpoint, all updates are guaranteed to commit to their final storage locations.

As the updates commit after the checkpoint, our schemes do not force disk write serialization across different block addresses. Instead, the data blocks buffered in the Memory Buffer (or mappings in the NVRAM) are written to the final disk location in an overlapped manner. They can even proceed with some re-ordering. Overlapping and re-ordering enables higher performance without affecting correctness. In theory, the performance could be even higher than a system without recovery.

We have seen that in all cases, if a client receives a transaction-completion message (Figure 2.4), then disk updates are guaranteed to commit. Still, it is possible that the client receives the completion message and sends another request before the server has physically finished the disk writes. Even if this request needs to read the data that is still being written to disk, no race occurs. The reason is that the PDD will automatically redirect the request to the Memory Buffer, which only deallocates an entry when the final disk update is completed.

2.3.3 Overheads and Recovery Latency

Overheads

ReViveI/O increases the latency of network messages because the server PDD does not send packets until after the next checkpoint. In practice, back-end database servers running TPC-C class applications are not particularly latency bound. Adding tens of ms to each transaction to support recovery is tolerable.

However, we tune TCP in two ways. First, since packets now take longer to be acknowledged, we increase the sliding window [77] that buffers yet-to-be-acknowledged packets. There is no danger of buffer overflow because TCP throttles packet sending as a buffer becomes full. Second, since the server PDD sends packets after checkpoints, the packet round trip time becomes more variable. This additional variability disrupts TCP's flow control mechanism. To solve this problem, the server PDD does not send all the packets as fast as it can after a checkpoint; instead, it smooths out the traffic.

ReViveI/O's impact on computation, memory, and disk accesses may also affect application throughput. Consider computation and memory accesses. For each output request, ReViveI/O

requires an initial write and a later read to the Memory Buffer (Figure 2.4). For network I/O, the data written/read is only a pointer to the socket buffer; the actual packet payload is retained in the sliding window elsewhere in memory. For disk I/O, the data written/read is the block data and metadata (*Buffer*), or the mapping only (*Rename*). For *Rename*, the mapping is also written to the NVRAM at checkpoints. In addition, the PDD executes bookkeeping code to manage the Memory Buffer.

For disk accesses, *Buffer* performs two disk writes per write request, although only one is in the critical path (Figure 2.4-(b)). The one in the critical path is fast because the disk buffer is written sequentially, and it can be a small, fast disk. *Rename* performs two writes on the same disk per request: one for the data and one for the new mapping (Figure 2.4-(c)). The first one is in the critical path; the second one can be merged with other updates of mappings from the same checkpoint interval. In addition, *Rename* may require periodic disk block compaction.

Recovery Latency

The latency of a recovery depends on the type of fault. Consider first an MR fault. Figure 2.5 shows a ReVive recovery time-line from [61], for the worst MR fault: the permanent loss of one node at a checkpoint. The latency numbers assume a 100 ms checkpoint interval. On top of the ReVive recovery, the thick up-arrows show the three actions performed by ReViveI/O.

Immediately after fault detection, ReViveI/O resets the I/O devices, namely network card and disk. This operation kills any ongoing DMA, which could overwrite the data being restored in the rollback. This operation is quick — 1 ms or less. It is also device dependent: it involves writing to a special I/O port to reset the network card, and sending a signal to reset the disk controller. It does not require any slow disk access.

After the memory state has been rolled back to the checkpoint, ReViveI/O re-initializes the device drivers (Figure 2.5). They have been left in an inconsistent state relative to the reset devices. This operation involves updating data structures in memory such as buffers and pointers, and bringing back the device driver's configuration parameters. It typically takes 10 ms or less.

Finally, after application execution has resumed, ReViveI/O performs in the background all the output operations needed to bring the I/O state to the checkpoint immediately preceding the fault.

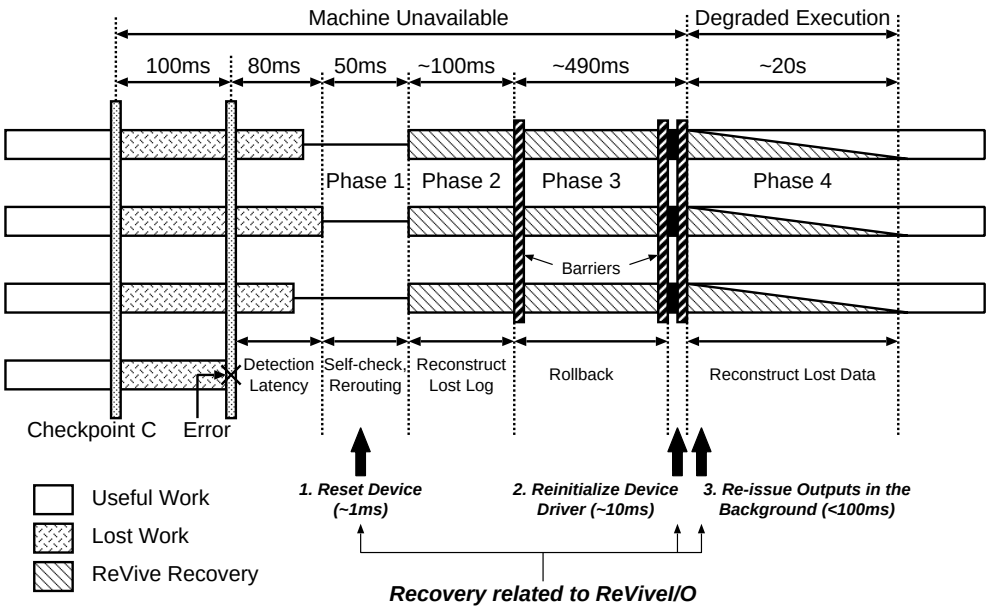

Figure 2.5: Recovery time-line for the permanent loss of one node at a checkpoint. ReViveI/O actions are shown with thick arrows.

Performing all these operations is easy. Indeed, for *Buffer*, the rolled-back Memory Buffer has an accurate record of all such operations. For *Rename*, the Memory Buffer has the record for the network operations, while the NVRAM has the record of the disk mappings to save. Performing these actions degrades the machine's performance for tens of ms, but does not make it unavailable.

Overall, the three ReViveI/O-related recovery actions negligibly add to the unavailable time and keep the fault transparent to the database.

Consider now an NMR fault. All currently-executing transactions abort and the system is typically rebooted. Before the database can use its own recovery mechanisms, ReViveI/O brings the disk to the state at the checkpoint immediately preceding the fault. This is done as follows: for *Buffer*, the disk output information is recovered from the disk buffer area; for *Rename*, the disk mappings are recovered from the NVRAM. The latency of performing these actions is much smaller than the time required for rebooting or for the database to recover. Therefore, ReViveI/O again adds negligibly to the unavailable time.

Finally, a second fault may occur while recovering. If ReVive can recover the memory state, ReViveI/O re-executes the three actions shown in Figure 2.5. Otherwise, ReViveI/O brings the

disk to a consistent state after reboot as just described.

2.3.4 Limitations

ReViveI/O has several applicability limitations. First, it is not applicable to latency-critical workloads, such as those with user interaction through graphics, keyboards, or other devices.

ReViveI/O relies on system-level code to intercept I/O requests, buffer them, and perform the operations later. This approach rules out, as they are currently implemented, user-level I/O and I/O co-processors such as TCP Offload Engines (TOEs). User-level I/O relies on user libraries to perform I/O, eliminating kernel involvement. For example, uncached accesses from user mode to the network interface send messages without involving the kernel. To be able to support schemes similar to ReViveI/O, we would have to add a PDD component to the user libraries

TOEs implement TCP operations in hardware. The kernel is not involved in performing the low-level operations in packet handling. Again, to support schemes similar to ReViveI/O, we would have to modify the TOE hardware to perform the PDD operation, namely buffer the packet for later issue. We would also have to synchronize the processor and the TOE at checkpoints.

We feel that, while user-level I/O and TOEs are interesting alternatives, they are still new technologies with poor standardization and, as a result, are hard to maintain in large server installations. The standard software TCP solution that ReViveI/O supports is overwhelmingly the most popular one.

2.4 Implementation Aspects

We have implemented a ReViveI/O prototype on a multiprocessor server with two 1.5 GHz AMD Athlon processors, 1.2 Gbytes of memory, two 80-Gbyte IDE disks, and a 1 Gbit ethernet card. One of the two disks is used as a disk buffer. The server runs Linux 2.4. The PDD is about 2,000 lines of C code for the disk and 2,000 for the network. The DDs, the Linux kernel, and the applications remain unmodified.

Note that our server does not have ReVive hardware. Consequently, when needed, we simulate its effect. Specifically, to test recovery, we pretend that a fault occurs immediately after a checkpoint

and, therefore, the memory state rolls back instantly. ReViveI/O can then proceed with resetting the devices, re-initializing the DDs and issuing all the buffered outputs. Since the memory state recovery and the I/O state recovery are conveniently decoupled (Figure 2.5), we can test the correctness of the I/O-related recovery without memory-checkpointing hardware. Under fault-free conditions, we do not model ReVive. However, in Section 2.6.3, we estimate the combined overhead and availability of ReVive and ReViveI/O.

Except for the ReVive support, we have thoroughly tested the prototype under many workload conditions (e.g., heavy disk writes, frequent small messages, or bulk data transmission) and restart scenarios (e.g., DMA in progress or many pending I/O requests). We also injected different faults that allowed us to test most software paths. In the following, we outline some implementation aspects.

2.4.1 Support for Disk I/O

ReViveI/O can be designed for disks accessed through a file system or as raw devices. In our prototype, we use a file system. Figure 2.6 expands Figure 2.2 showing the interface between kernel, PDD, and disk DD [7] for *Buffer* and *Rename*. The modules in shaded pattern are those added for ReViveI/O: PDD, Memory Buffer, disk buffer or rename area, and NVRAM.

In a conventional system, a read request causes a buffer cache access. If a miss occurs, the low-level DD satisfies the request. In a write, a block is allocated in the buffer cache if it is not already there. The block is updated and marked dirty. Sometime later, the kernel writes it to disk.

With ReViveI/O, such dirty block writes are directed to the PDD. As indicated before, the PDD buffers the information and commits it after the next checkpoint. Although individual DMA operations (e.g., setup, execution, postprocessing) performed by the low-level DD may not be idempotent, disk writes at the PDD level are idempotent because the PDD triggers these low-level operations as an indivisible operation.

The Memory Buffer is sized based on the machine's total disk bandwidth and the checkpoint interval. The same applies to the NVRAM except that the bandwidth is per disk, since ReViveI/O has one NVRAM per disk. For example, a 100 MB/s disk array in a 100 ms checkpoint interval can consume 10 MB. Consequently, for *Buffer*, a 20 MB Memory Buffer and a 20 MB disk buffer

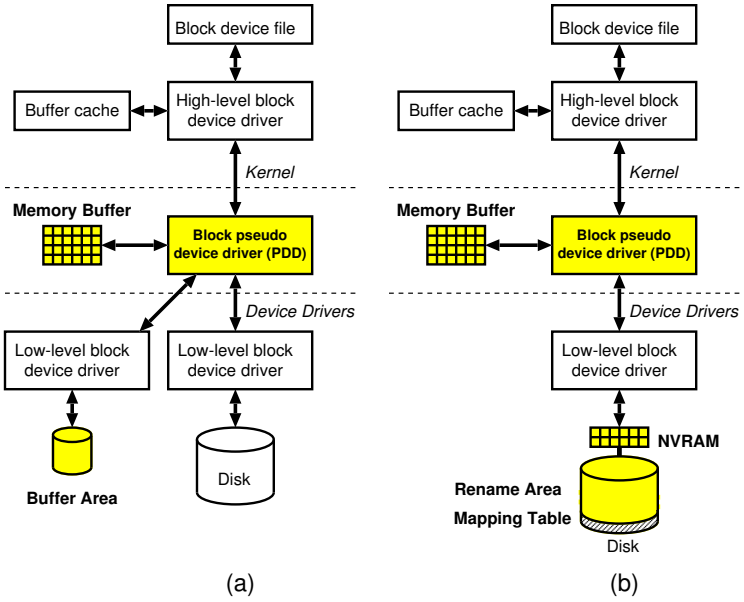

Figure 2.6: Interface between kernel, PDD, and disk DD for *Buffer* (a) and *Rename* (b).

area suffice. For this update rate, *Rename* generates about 20 KB of mappings per checkpoint interval. Consequently, for *Rename*, a 20 MB disk rename area, 40 KB Memory Buffer, and 40 KB NVRAM suffice. Since the checkpoint operation takes about 1 ms [61], an NVRAM built out of battery-backed SRAM has sufficient bandwidth (∼100 MB/s [76]) to load these 20 KB mappings during a checkpoint.

2.4.2 Support for Network I/O

The kernel does not use the usual interface (e.g., `eth0`). Instead, it uses the virtual interface provided by the network PDD (say, `veth0`). The data structure passed between the kernel and the network DD is the socket buffer, which contains the length of the packet, a pointer to the packet, and other fields. When the kernel passes a socket buffer to the PDD, a pointer to it is copied to the Memory Buffer. We copy only the pointer to reduce overhead. After the checkpoint, the socket buffer is passed to the network DD.

When an input packet arrives at the network card, the appropriate handler is triggered in the network DD, which in turn calls the `netif_rx` function in the kernel to process the packet. TCP

Workload		Description	I/O
Micro-Benchmarks	RandomWrite	Repeatedly write blocks of a given size to disk. The writes are synchronous and directed to random locations. Size can be set.	Disk
	SequentialWrite	Like RandomWrite but writes are directed to sequential locations.	Disk
	Iperf [32]	Repeatedly send messages of a given size. Size can be set.	Network
Throughput oriented	TPC-C-like on Oracle 9.0.2	32 warehouses, 30 remote clients, no think time, 400-Mbyte database buffer, and 4-Kbyte blocks	Disk and network
Latency bound	WebStone 2.5 with Apache 2.0	Memory resident, variable number of remote clients, no think time, 85 HTML documents of 100 KB on average	Network

Table 2.1: Workloads used in the evaluation. We use the term "TPC-C-like" because compliance with the specification is not fully checked.

would get confused if the kernel sent a packet through `veth0` and received the reply from `eth0`. Consequently, the DD call is routed through a `netif_rx` function in a special library that changes the device field of the socket buffer to `veth0`. Neither DD nor kernel are modified.

2.5 Evaluation Methodology

We evaluate the three schemes of Table 2.2. Of the ReViveI/O approaches, we select *Buffer* for evaluation. *NoRollback* is the unmodified server, which has no provision for I/O undo/redo. *Stall* is a scheme for disk I/O similar to Masubuchi *et al.* [47, 48]. In *Stall*, there is no data buffering; requesting processes are not notified of output I/O completion until the next checkpoint.

Scheme	Description
Buffer	ReViveI/O approach. Supports I/O undo/redo for disk & network I/O.
NoRollback	Unmodified server. No provision for I/O undo/redo.
Stall	Output I/O blocks until next checkpoint. Scheme for disk I/O only. Similar to Masubuchi *et al.* [47, 48].

Table 2.2: Schemes evaluated.

We run the workloads of Table 2.1: a throughput-oriented one (TPC-C on Oracle 9.0.2), a latency-bound one (WebStone [83] on the Apache server [1]), and several microbenchmarks.

We experiment with 20-240 ms checkpoint intervals. For each checkpoint interval, we set the TCP sliding window size to buffer all unacknowledged packets at the 1 Gbit ethernet bandwidth, and the Memory Buffer size to hold all the data that can be written to disk. For 80 ms intervals,

this is 12 MB for the sliding window and 8 MB for the buffer. The ratio is the same for the other intervals.

2.6 Evaluation

To evaluate our ReViveI/O prototype, we measure its overhead in fault-free execution (Section 2.6.1) and the latency of fault recovery (Section 2.6.2). Then, we project the impact of combining ReViveI/O and ReVive (Section 2.6.3).

2.6.1 Execution Overhead

Disk I/O Microbenchmarks

The RandomWrite and SequentialWrite microbenchmarks test worst-case disk I/O conditions. They consist of a loop that synchronously writes blocks of a given size to disk. Consequently, the disk is constantly busy. In our server, one disk can support up to 32 Mbytes/s of write throughput, while the other (used as disk buffer) up to 36 Mbytes/s. With such hardware, Figure 2.7 shows the resulting system throughput as a function of the size of the blocks written. We consider 20 and 160 ms checkpoint intervals.

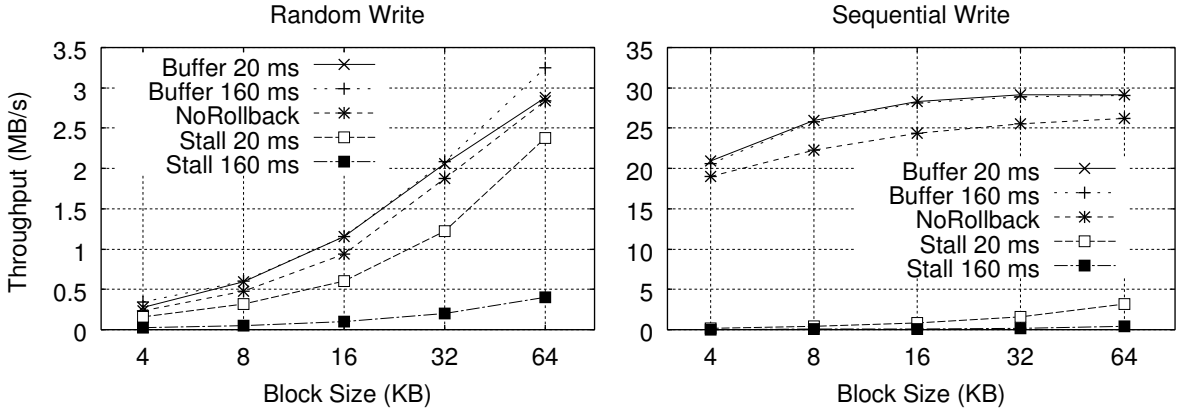

Figure 2.7: Disk I/O throughput for random and sequential synchronous writes.

We see that the various overheads of *Buffer* (Section 2.3.3) do not reduce throughput relative to *NoRollback* under heavy disk write traffic. In fact, *Buffer*'s throughput is slightly higher than

NoRollback's. The reason is that a write request in *Buffer* returns as soon as the data is written to the faster disk *sequentially*, rather than to the other disk randomly (RandomWrite) or sequentially (SequentialWrite).

The checkpoint interval size has little effect on *Buffer*. The reason is that PDD operations have only tiny overhead. Finally, *Stall* delivers a very low throughput; effectively, *Stall* manages only a single synchronous write per thread per checkpoint.

Network I/O Microbenchmark

The Iperf microbenchmark measures the maximum TCP bandwidth. Figure 2.8 shows the sustained throughput as a function of the message size for two cases: our server sends messages to one client (Unidirectional) and both client and server send messages to each other (Bidirectional). The case where the client sends messages to the server is similar to Unidirectional — messages have the same Round Trip Time (RTT) because the server delays all packets, including ACKs. In this experiment, *Stall* does not apply.

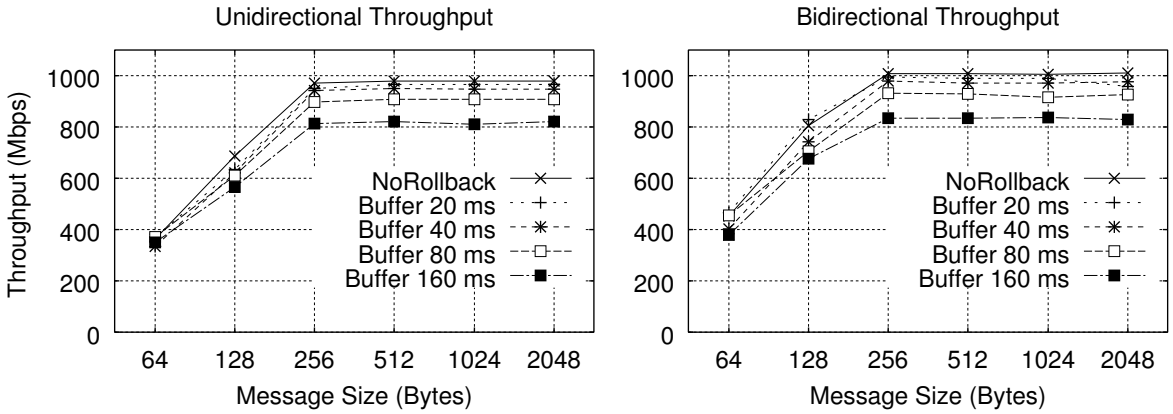

Figure 2.8: Uni- and bi-directional throughput between one client and the server over 1 Gbit ethernet.

We see that, under these extreme conditions, *Buffer* lowers the throughput relative to *NoRollback*. This is due to PDD overheads (Section 2.3.3) and suboptimal TCP operation in a high-bandwidth, high-latency network. However, the throughput reduction is modest: it ranges from 5% with a 20 ms interval to less than 20% with 160 ms.

Finally, *Buffer* also increases packet RTT. The resulting impact on the response time depends

on the application. The impact is tolerable in applications where the server performs substantial work or where the communication pattern involves bulk data transmission. We consider this issue next.

Throughput-Oriented Workload: TPC-C + Oracle

This workload is typical of back-end database servers. The major concern is not response time, but maintaining high throughput. Individual transactions can take significant time, as they typically perform disk I/O in the server. In *Buffer*, this workload exercises both disk and network PDDs.

Figure 2.9 shows the average TPC-C throughput with different schemes and checkpoint intervals normalized to that in *NoRollback*[2]. The throughput of our moderately tuned *NoRollback* setup is 1561 transactions per minute and its average response time is 612 ms. The figure shows data for *Stall*, and for *Buffer* with disk PDD only and with disk plus network PDDs. To minimize errors, we report conditions of the first 10-minute interval after the database is warmed up and the throughput becomes steady. During that period, we take a measurement every 30 seconds. The figure shows the mean and standard deviation of such measurements.

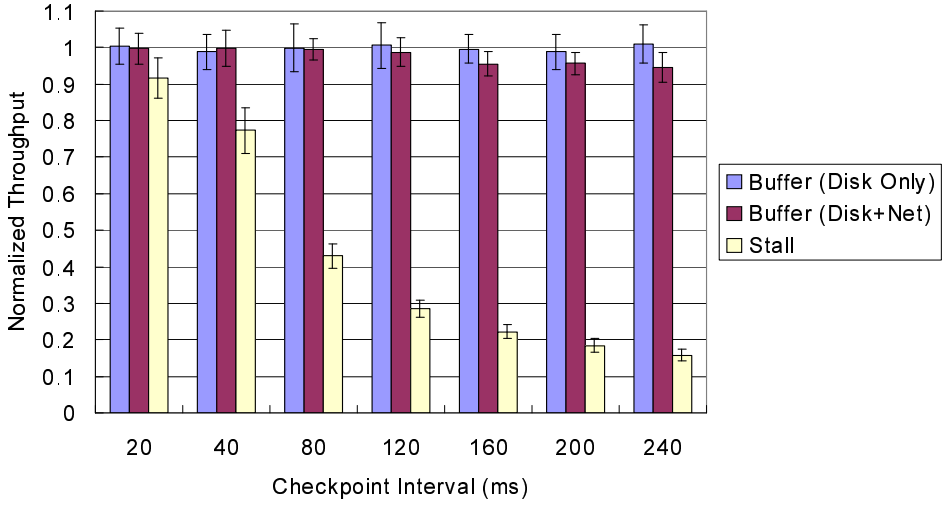

Figure 2.9: Transaction throughput with different schemes and checkpoint intervals normalized to *NoRollback*. The experiments run with 30 remote clients.

Consider *Stall* first. If we can only tolerate a 5% reduction in throughput, none of the checkpoint

[2]Throughput is given in New Order transactions per minute; response time is for New Order transactions as well.

intervals shown is acceptable. In contrast, *Buffer* keeps the throughput reduction within 1% up to 120 ms checkpoint intervals. Note that for checkpoint intervals above 120 ms, the throughput reduction comes mainly from network PDD overhead. Interestingly, the average response time does not degrade as we increase the checkpoint interval. In fact, it goes down slightly, decreasing to 590 ms by the time we use 240 ms checkpoint interval. The reason is that long intervals reduce the transaction rate, which in turn diminishes disk contention.

We have repeated the *Buffer* experiments for different numbers of clients and obtained similar results. Figure 2.10 shows the normalized throughput for a range of checkpoint intervals for 60 and 90 clients. Each bar is normalized to *NoRollback* for the same number of clients. As we increase the number of clients (i.e., more transactions overlap in time), the delay incurred by the network PDD is less visible and, therefore, long checkpoint intervals become more tolerable.

Overall, we conclude that in throughput-oriented workloads like TPC-C, and for 20–120 ms checkpoint intervals, our proposed *Buffer* scheme induces very small throughput reductions of up to 1%. In such workloads, the progress of transaction processing typically depends on the rate of synchronous writes issued by the database log-writer process. Our *Buffer* scheme affects such progress minimally.

Latency-Bound Workload: WebStone + Apache

In this workload, multiple remote clients read HTML documents that are memory-resident in the server. Each transaction involves establishing the connection with a three-way handshake, reading a file, and closing the connection. Transactions are short because there is no disk I/O — only the network PDD is exercised. This workload simulates an interactive environment. Consequently, we are interested in response time, measured as the time between requesting the connection until the whole file is received.

Figure 2.11 shows the response time for different numbers of clients and checkpoint intervals. The figure is organized in numbers of clients. In each group, there are bars for *NoRollback*, and for *Buffer* with different checkpoint intervals. Since there is no disk I/O, the *Stall* scheme is irrelevant. To obtain the data, we run each experiment for 10 minutes, with the server at 100% CPU usage.

We see that the response time quickly increases with the checkpoint interval. With a checkpoint

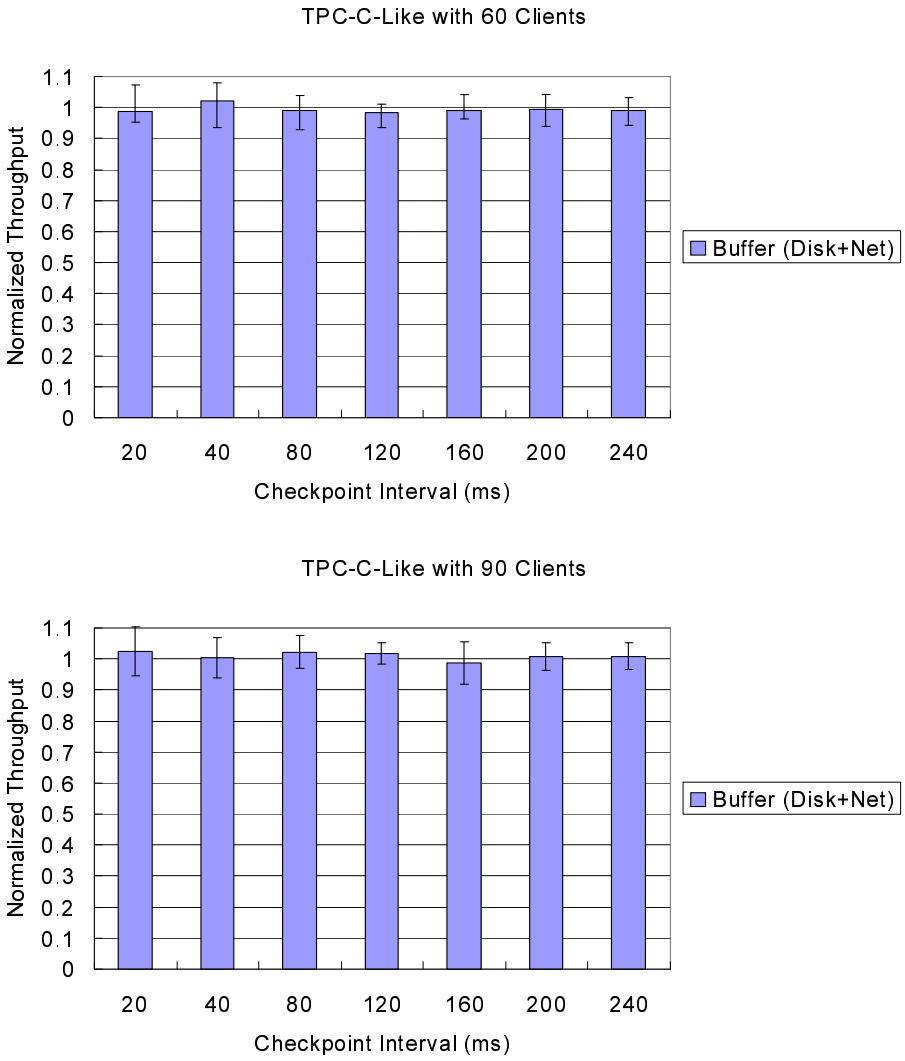

Figure 2.10: Normalized transaction throughput as a function of the checkpoint interval for different numbers of clients.

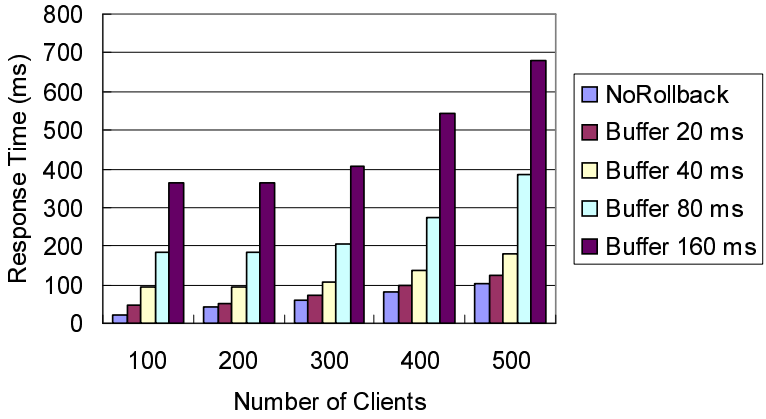

Figure 2.11: Response time for different numbers of clients and checkpoint intervals.

interval T, the response time should increase by $2 \times T$, since we add T to establish the connection and T to get the data (Recall from Section 2.3.3 that the PDD smooths out outgoing messages). This is what we observe with 100 clients. For more than 300 clients and $T \geq 80$ms, contention causes larger increases in the response time.

According to [56], it is acceptable to add up to 100 ms to the response time of a transaction. Consequently, our *Buffer* scheme can be used in the web server measured, as long as the checkpoint interval is ∼50 ms or shorter.

2.6.2 Latency of Fault Recovery

To recover from an MR fault, our schemes perform three actions (Section 2.3.3): reset the devices, re-initialize the DDs and, in the background, perform all the buffered output operations to bring the I/O state to the checkpoint immediately preceding the fault. While our prototype server lacks ReVive hardware, we can measure the recovery latency of ReViveI/O as discussed in Section 2.4.

We have measured the latency of each part of the recovery for *Buffer*, and listed average values in Table 2.3. From the table, we see that device reset and DD re-initialization are quick. Note that, to re-initialize the disk DD, we do not need to access the disk to get information such as the number of cylinders and the sector size; these parameters are obtained from the recovered memory. Finally, the third operation takes tens of ms, but it is executed in the background. Overall, compared to the ReVive recovery latency (Figure 2.5), ReViveI/O adds negligible recovery overhead.

Operation	Duration	
	Disk	Network
Reset device	1 ms	15 μs
Re-initialize device driver	10 ms	60 μs
Re-issue operations in background	$\sim T$	$\sim T$

Table 2.3: Latencies of the operations needed to recover from an MR fault. T is the checkpoint interval.

If the fault is NMR, *Buffer* only needs to re-issue the buffered output operations. This activity takes several seconds, as *Buffer* has to read data and mappings from the disk buffer. Such latency is negligible compared to the tens of minutes needed to reboot the server and run a database recovery routine after this type of fault.

2.6.3 Combining ReVive and ReViveI/O: Performance Overheads and Availability

We would like to estimate the impact of combining ReViveI/O and ReVive. In [61], ReVive was evaluated for a checkpoint interval $T = 100$ ms, where each checkpoint took 1 ms. ReVive induced a 6% execution overhead.

We model ReVive as inducing a $c = 1$ ms overhead every checkpoint, and a fixed $r = 5\%$ overhead for the period between checkpoints, independently of T. Therefore, the throughput reduction factor induced by ReVive is $f = \frac{c+(T-c)\times r}{T}$. The response time increase due to ReVive is $t_s \times \frac{f}{1-f}$, where t_s is the time that the transaction spends executing in the server.

Figure 2.12 takes the impact of *Buffer* on TPC-C throughput (Section 2.6.1) and WebStone response time (Section 2), and adds the estimated effect of ReVive. We can see that a throughput-oriented workload such as TPC-C keeps the throughput reduction at 7% or below for checkpoint intervals between 60 and 120 ms. Most of the overhead is due to ReVive. On the other hand, the response time increase in a latency-bound workload such as WebStone is practically all due to ReViveI/O. The increase is $2 \times T$, where T is the checkpoint interval (Section 2).

To complete the picture, we compare the availability of *ReVive+Buffer* and *NoRollback*. For the former, we assume that MR and NMR faults are independent and distributed exponentially. As

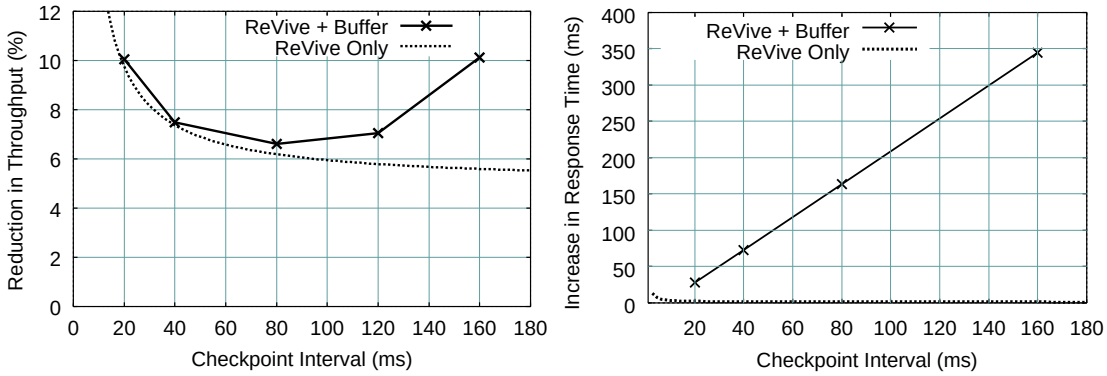

Figure 2.12: Estimated combined effect of ReVive and ReViveI/O on the throughput of TPC-C (left) and the response time of WebStone (right).

a result, the availability of *ReVive+Buffer* is $1 - \frac{MTTR_{MR}}{MTBF_{MR}} - \frac{MTTR_{NMR}}{MTBF_{NMR}}$. We estimate $MTTR_{NMR}$ as 5 minutes for machine reboot plus 5 minutes for database recovery [49]. We set $MTTR_{MR}$ to 1 second [61]. For *NoRollback*, all faults have the same MTTR, namely $MTTR_{NMR}$.

Figure 2.13 shows the unavailability as a function of $MTBF_{MR}$. Note that both axes in the figure are logarithmic. For *ReVive+Buffer*, we show two curves: 1:100 assumes that $MTBF_{NMR} = 100 \times MTBF_{MR}$, while 1:1000 assumes that $MTBF_{NMR} = 1000 \times MTBF_{MR}$. For *NoRollback*, both curves are practically the same, and we show only one.

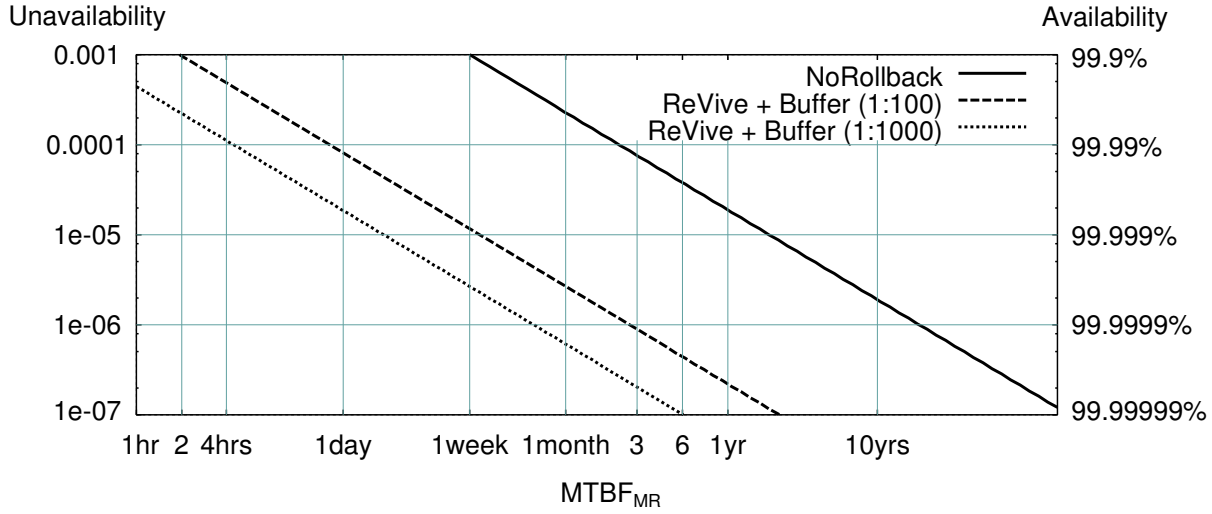

Figure 2.13: Unavailability as a function of the MTBF of MR faults. The MTBF of NMR faults is set to 100 or 1000 times the MTBF of MR faults (curves 1:100 and 1:1000, respectively).

The figure shows that *ReVive+Buffer* has much lower unavailability than *NoRollback* thanks

to its tiny recovery latency for the more common MR faults. For example, for 1-week $MTBF_{MR}$, *ReVive+Buffer* (1:100) has an unavailability of $\sim 10^{-5}$, which corresponds to 99.999% availability, while *NoRollback* has an unavailability of 0.001, which corresponds to 99.9% availability. Overall, *ReVive+Buffer*'s unavailability is 86 and 375 times lower than *NoRollback*'s for 1:100 and 1:1000, respectively.

In summary, *ReVive+Buffer* provides higher availability than conventional systems while delivering slightly lower throughput. We believe that, at least in the applications considered, reducing downtime is much more important than achieving peak throughput while the machine is up.

Chapter 3

Architecture-Aware Fine-Grain Body Biasing for Within-Die Parameter Variation

3.1 Introduction to Architecture-Aware Fine-Grain Body Biasing

As high-performance processors move into sub-65 nm technologies, designers face the major roadblock of parameter variation — the deviation of Process, Voltage, and Temperature (PVT [6]) values from nominal specifications. Variation makes designing processors harder because they have to work under a range of parameter values.

Variation is induced by several fundamental effects. Process variation is caused by the inability to precisely control the fabrication process at small-feature technologies. It is a combination of systematic effects [20, 58, 74], e.g., due to lithographic lens aberrations, and random effects [5], e.g., due to dopant density fluctuations. Voltage variation is mostly due to IR drops resulting from non-ideal voltage distribution. Finally, temperature variation is largely due to different level of activity across the chip. All these variations become harder to tolerate as technology scales to minute feature sizes.

A key process parameter subject to variation is the transistor threshold voltage (V_{th}). V_{th} is especially important because it directly impacts major architectural characteristics of the chip. Specifically, V_{th} variations affect the speed and leakage power of a transistor [78] and, in aggregate form, of the whole chip. Moreover, V_{th} is also a function of temperature, which increases its variability.

A recently-proposed technique to mitigate V_{th} variation within a chip is Fine-Grain Body Biasing (FGBB) [81]. FGBB applies different body biases to different sections of the chip, which we call *Cells*. A body bias is a voltage applied between the source or drain of a transistor and its

substrate, effectively changing the transistor's V_{th} [78]. Depending on the polarity of the voltage applied, V_{th} increases or decreases. If it increases, a leaky transistor becomes less leaky but slower; if it decreases, a slow transistor becomes faster but leakier. Body Biasing (BB) had already been applied chip-wide to increase a chip's frequency, reduce its leakage power, or for other purposes (e.g., [40, 50, 81]). However, Tschanz *et al.* [81] are the first to apply it to multiple sections of the chip separately, to improve its effectiveness in the presence of within-die V_{th} variation.

In Tschanz *et al.*'s design, the chip has 21 cells. They are rectangular slabs abutting each other. The authors reduce the V_{th} of slow cells and increase the V_{th} of leaky cells. The result is a chip with improved frequency-leakage operation. However, there is no justification of the cells' shape or number.

In this study, we claim that there are major advantages to using *architecture awareness* in selecting the cells in FGBB. Cells should mostly follow the shapes of architectural modules, such as execution units or caches — an approach that we term *Architecture-Aware FGBB (AA-FGBB)*. The rationale is that architectural functionality is the main determinant of the *temperature* and the *type of critical path* (memory versus logic) in a section of the chip; these characteristics in turn determine the optimal BB to apply in that section to mitigate variation.

The main contribution of this work is to introduce and demonstrate this new idea. A second contribution is the development of a novel, parameterized model of V_{th} variation within the chip. In the process, this work shows that computer architecture plays an essential role in mitigating parameter variation in small-feature technology.

We present results based on simulating batches of 200 chips with variation. AA-FGBB enables 35% of the chips to work at the highest frequency, compared to only 18% with advanced conventional FGBB — potentially increasing each improved chip's value by 50%. AA-FGBB also reduces the leakage of the chips by 40%, compared to 25% with advanced conventional FGBB. Finally, compared to the original batch of chips, AA-FGBB enables 82% of the chips to work at the top two frequencies rather than only 14%, potentially increasing each improved chip's value two or three times.

3.2 Background

Variation can be die-to-die (D2D) or within die (WID). We focus on the latter, which offers more challenges and opportunities for computer architecture solutions. In addition, in the context of BB, D2D V_{th} variation can be treated as an offset to WID variation. We consider temperature (T) variation. However, due to the complexity of the problem, we do not consider voltage (V) variation in this study. Next, we review transistor leakage, gate delay, and BB.

3.2.1 Transistor Leakage and Gate Delay

Transistor leakage and gate delay are a function of two key varying parameters, namely V_{th} and T. Specifically, a transistor's subthreshold leakage current is:

$$I_{\text{leak}} \propto T^2 e^{-qV_{\text{th}}(T)/kT} \tag{3.1}$$

where V_{th} decreases by 2.5 mV/K as temperature increases [39]. From the equation, as V_{th} decreases, leakage goes up. Moreover, as T increases, leakage quickly goes up, both because of its dependence on T and because V_{th} goes down. Figure 3.1 shows the dependence of I_{leak} on V_{th} and T, where the leakage current is normalized by I_{leak} at $V_{\text{th}} = 150$ mV and 100 °C. From this plot, it is clear that the temperature variation has more impact on the leakage than the process variation.

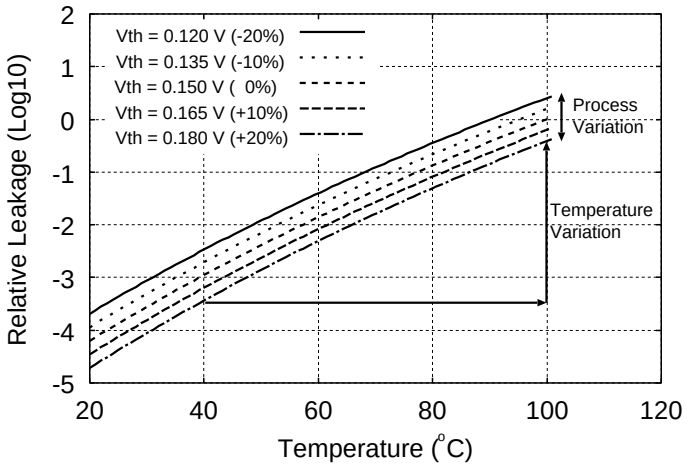

Figure 3.1: Temperature dependence of the subthreshold leakage current.

The delay of an inverter gate is given by the alpha-power model [67] as:

$$T_g \propto \frac{LV}{\mu(T)(V - V_{\text{th}}(T))^{\alpha}} \tag{3.2}$$

where α is typically 1.3 and μ is the mobility of carriers ($\mu(T) \propto T^{-1.5}$). As V_{th} decreases, $V - V_{\text{th}}$ increases and the gate becomes faster. As T increases, $V - V_{\text{th}}(T)$ increases, but $\mu(T)$ decreases [39]. Figure 3.2 plots the switching frequency for various temperature and V_{th}. Contrary to the leakage, the frequency is more affected by the process variation than the temperature variation.

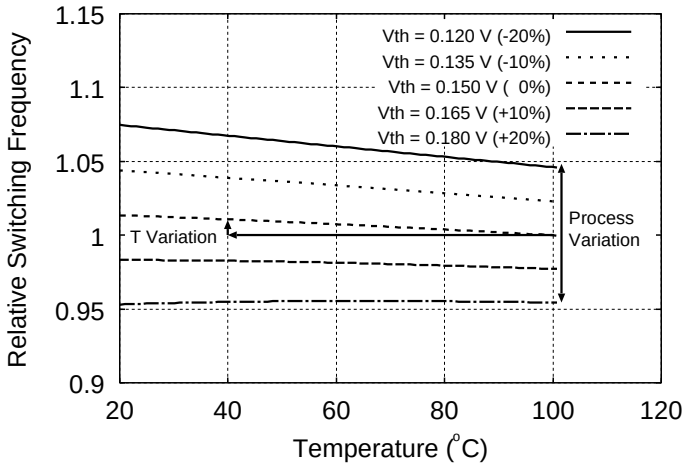

Figure 3.2: Temperature dependence of the switching frequency.

3.2.2 Body Biasing

Body Biasing (BB) a transistor involves applying a voltage between its source or drain and substrate to alter the subthreshold leakage current [78]. In Forward BB (FBB), the voltage polarity is such that V_{th} decreases. The result is a faster and leakier transistor. In Reverse BB (RBB), the voltage polarity is such that V_{th} increases, creating a slower, less leaky transistor. These properties make BB a useful architecture knob, although it has been little explored.

BB's static application does not have any noticeable performance overhead [54]. It adds certain complexity to the design, as triple wells are required [43] and wires need to be added to transmit the bias current [54]. The area overhead is modest [54], as we discuss next. Overall, the complexity and cost are manageable, and many papers have recently appeared with circuits benefiting from

FBB, RBB, or both (e.g., [40, 50]).

Fine-Grain Body Biasing (FGBB) is our term to refer to applying different BB to different sections of the chip. We call such sections *Cells*. The only published work on FGBB is that of Tschanz *et al.* [81], who describe an experimental chip with 21 rectangular cells. They use it to mitigate WID $V_{\rm th}$ variation: cells with slow transistors are made faster with FBB; cells with leaky ones are made less leaky with RBB.

Uses of BB and FGBB. While no commercial processor uses BB (to our knowledge), there are several proposed uses. One use of BB is to save dynamic power [46, 54]. By applying FBB, we can reduce $V_{\rm th}$, which in turn enables a reduction of V for the same chip frequency (Eq. (3.2)). For a range of FBB, the result is a dynamic power reduction that is larger than the accompanying increase in leakage power, thereby reducing total power.

A second use of BB is to reduce Die-to-Die (D2D) process variation [6, 81]. Different dies from the same wafer run at different frequencies and leak a different amount. Applying chip-wide BB can help push chips into acceptable ranges of frequency and leakage. Specifically, RBB is applied to fast and high-leakage chips to reduce leakage; FBB is applied to slow and low-leakage chips to increase speed.

Finally, as indicated above, Tschanz *et al.* [81] use FGBB to reduce WID variability. The result is an even better frequency-leakage operation of the chip.

This study builds on Tschanz's work. We apply a computer architecture perspective to FGBB, arguing that FGBB should be architecture-aware. Also, we present a new model of WID $V_{\rm th}$ variation and analyze FGBB's impact in depth.

Circuitry for BB and FGBB. BB requires enhancing the manufacturing process with a triple well process [43], adding power lines for the body bias voltage, and adding critical-path circuits to calibrate the BB [54, 81]. The area overhead of the power lines vary a lot, depending on layout conditions. Such overhead can be up to 8% of the area in hard-to-route areas of control logic, but it is much smaller in memory areas [68]. For the chip as a whole, the area overhead is modest. Perhaps as a result, BB papers do not dwell on such overhead [54, 81]. Instead, they focus on the overhead of the calibration circuits. Each cell has at least one circuit that is representative of the critical paths in the cell. Using a mechanism similar to Razor [18] on that representative

circuit, [54, 81] determine the frequency that the transistors in that cell can support (Figure 3.3). Based on it, they can determine the desirable static BB to apply to the cell. We note that each cell in Tschanz *et al.* [81] only has a single critical path; consequently, the sample used to calibrate the cell is trivially representative of the cell. In their experimental chips, [54, 81] report a 2% chip area overhead for all these circuits at 150nm. An optimized design of these circuits in a high-volume processor chip at 45nm technology (which we are targeting) will have little area overhead.

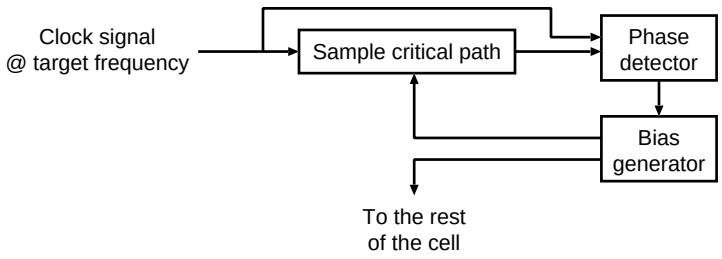

Figure 3.3: Block diagram of a circuitry that adjusts body bias.

3.3 WID V_{th} Variation and its Impact

From an architectural perspective, $V_{\rm th}$ variation is of key importance: it directly affects a chip's leakage and frequency. The WID variation of $V_{\rm th}$ is impacted by both systematic and random effects [5]. Limitations of the lithography and other manufacturing processes introduce systematic variations in $V_{\rm th}$, which repeat across dies. Typically, such variations exhibit a spatial structure with a certain scale of parameter changes over the two-dimensional space [20, 58, 74]. On the other hand, a variety of materials effects, most notably changes in the dopant density of the channel [5], introduce random variations in $V_{\rm th}$. Such variations have a different profile in each die and are in effect noise superimposed on the systematic variation. Since there is no existing model of WID $V_{\rm th}$ variation, we develop a new one.

3.3.1 A Model of WID V_{th} Variation

Rather than building the model bottom-up, starting from largely unknown random and systematic $V_{\rm th}$ components, we build it top-down, relying on one accepted assumption and on experimental data from Friedberg *et al.* [20]. The resulting model does not explicitly separate systematic and

random effects. Instead, it is used by choosing values for three intuitive parameters: mean μ, standard deviation σ, and range ϕ.

The model treats V_{th} at position $\vec{x}$ in the chip as a random variable $V_{\text{th}}(\vec{x})$ that follows a normal distribution with mean μ and standard deviation σ, regardless of $\vec{x}$. This is consistent with current assumptions [19, 53]. To capture systematic variation, we use an appropriate spatial distribution of V_{th}. For that, we assume a correlation between $V_{\text{th}}(\vec{x})$ and $V_{\text{th}}(\vec{y})$ that depends only on the distance between $\vec{x}$ and $\vec{y}$, and not on the position or direction. We make this assumption of uniformity and isotropy in this initial work for simplicity. We can thus write the correlation function of $V_{\text{th}}(\vec{x})$ and $V_{\text{th}}(\vec{y})$ as $\rho(r)$, where $r = |\vec{x} - \vec{y}|$ (Appendix B.1). While $\rho(0) = 1$ (i.e., totally correlated) by definition, we set $\rho(\infty) = 0$ (i.e., totally uncorrelated) because two infinitely separated points have independent V_{th} when we only consider WID variation.

To determine how $\rho(r)$ changes from $\rho(0) = 1$ to $\rho(\infty) = 0$ as r increases, we use the Spherical model [14, 36]. This model is very similar to the correlation function experimentally measured by Friedberg *et al.* [20] for the WID variation of gate length. Our rationale is that gate length variation is the main determinant of systematic V_{th} variation. Figure 3.4 shows the chosen function $\rho(r)$, while Appendix B.1 gives its analytic expression. At a finite distance ϕ called *range*, the function converges to zero.

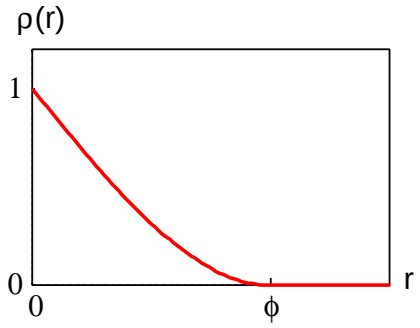

Figure 3.4: Correlation between the V_{th} at two points as a function of the distance r between them.

Intuitively, our assumptions imply that the V_{th} of a transistor is highly correlated to the V_{th} of those in its immediate vicinity. The correlation decreases linearly with distance at small distances. Then, it decreases more slowly. At distance ϕ, there is no significant correlation between two

transistors' V_{th}.

We will explore different values of ϕ and σ/μ. We give ϕ as a fraction of the chip's width. A high ϕ implies that large sections of the chip are correlated with each other; the opposite is true for a low ϕ. The experiments by Friedberg *et al.* [20] found $\phi \approx 0.5$.

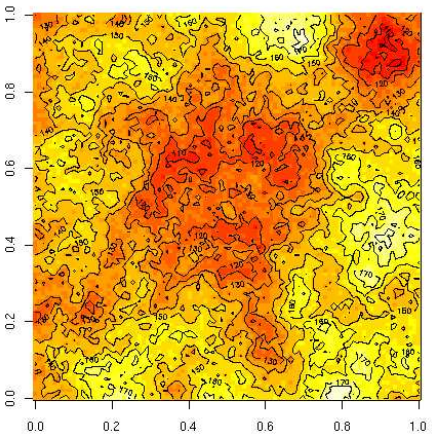

Figure 3.5: Randomly generated V_{th} distribution in a chip for $\phi = 0.5$.

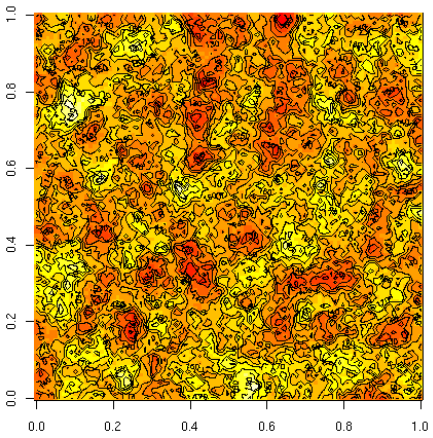

Figure 3.6: Randomly generated V_{th} distribution in a chip for $\phi = 0.1$.

As an illustration, Figures 3.5 and 3.6 show a chip's V_{th} distribution for $\phi = 0.5$ and $\phi = 0.1$, respectively, as generated by the geoR statistical package [64] of R [62]. We set $\sigma/\mu = 0.12$. In the $\phi = 0.5$ case, we discern large spatial features, whereas in the $\phi = 0.1$ one, the features are small. A distribution without any correlation ($\phi = 0$) contains only random variation and appears as white noise.

3.3.2 Impact on Chip-Level Behavior

Leakage Power

We take our V_{th} distribution and integrate the leakage equation (Eq. (3.1)) over all the transistors in the chip. The result is the total leakage current in the chip. Let us call P_{leak} and I_{leak} the chip leakage power and current under V_{th} variation, and P^0_{leak} and I^0_{leak} the same parameters when there is no V_{th} variation. Appendix B.1 shows that:

$$P_{\text{leak}}/P^0_{\text{leak}} = I_{\text{leak}}/I^0_{\text{leak}} = e^{(q\sigma/kT)^2/2} \tag{3.3}$$

which says that the increase in the chip's leakage power and current due to V_{th} variation depends on the standard deviation σ of V_{th}'s variation. Figure 3.7 plots the relative power as a function of σ. It increases rapidly as σ goes up.

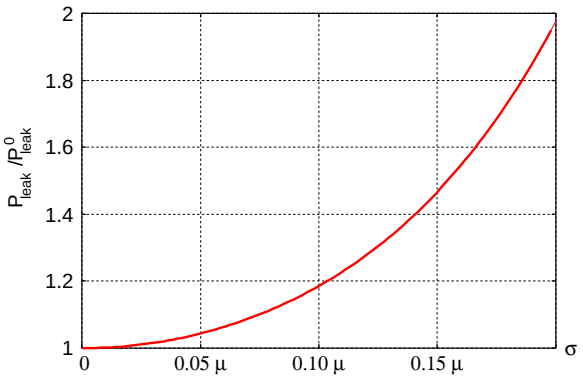

Figure 3.7: Relative leakage power in the chip as a function of V_{th}'s σ, for T=25 °C and μ=150mV.

Section 3.2.1 indicated that, as T goes up, leakage increases rapidly. Additionally, Eq. (3.3) shows that the ratio $P_{\text{leak}}/P^0_{\text{leak}}$ decreases with T. Variation is relatively less harmful at high T.

Chip Frequency

Given that V_{th} is normally distributed, Appendix B.1 shows that the gate delay of Eq. (3.2) is also normally distributed. Assuming that a critical path consists of n_{cp} gates, and that a modern processor chip has thousands of critical paths, Bowman *et al.* [8] compute the probability distribution of the longest critical path $\max\{T_{\text{cp}}\}$. This is the one that determines the processor frequency. Then, we set the chip frequency to be $1/\max\{T_{\text{cp}}\}$.

Using this approach, we find that V_{th}'s σ determines the chip frequency. Figure 3.8 shows the probability distribution of the chip frequency for different σ. The frequency is given relative to a processor without V_{th} variation (F/F_0). The figure shows that, as σ increases, (i) the mean chip frequency decreases and (ii) the chip frequency distribution gets more spread out. In other words, given a batch of chips, as σ increases, the mean frequency of the batch decreases and, at the same time, individual chip's frequency deviates more from the mean.

Appendix B.2 further develops the theory of D2D and WID variations under a constant tem-

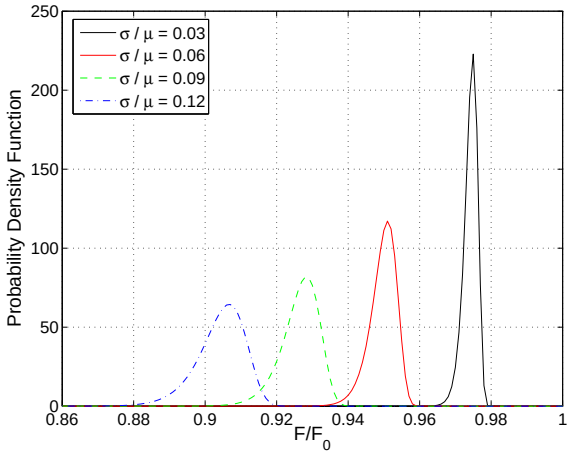

Figure 3.8: Probability distribution of the relative chip frequency as a function of V_{th}'s σ for T=25 °C, μ=150mV, 12 FO4s in the critical path, and 10,000 critical paths.

perature.

Summary

We see that V_{th}'s σ directly affects chip leakage and frequency. As σ increases, chip leakage increases rapidly, and chip frequency decreases in mean value and varies more. Therefore, V_{th} variation is detrimental.

3.4 Architecture-Aware Fine-Grain Body Biasing (AA-FGBB)

Judicious application of FGBB can redress the problem of WID V_{th} variation. With FGBB, the chip is divided into cells. Ignoring for the moment the effects of WID temperature variation, FGBB applies reverse BB to cells that have low V_{th}. FGBB applies forward BB to cells that have high V_{th}. The consequence is to lower the effective σ of V_{th}. Consequently, the resulting chip may increase its frequency, reduce its leakage, or a combination of both.

The only previous work on FGBB [81] constructed the BB cells as rectangular slabs abutting each other. Moreover, it applied BB to the chip at a constant temperature — therefore neglecting the effect of temperature variation across the chip under load. This study presents a new approach.

3.4.1 Rationale for Architecture Awareness

We claim that there are significant advantages to using *architecture-aware* BB cells — meaning cells with shapes that largely match architectural modules such as caches, registers, or execution units. We put forward two main reasons for this, namely (i) variations in temperature and (ii) differences in the types of critical paths in logic and memory modules.

Temperature Effects

Equations (3.1) and (3.2) show that transistor leakage and gate delay are significantly affected by temperature. At high temperature, transistors become vastly leakier and gates slower. Therefore, it is more effective to construct BB cells composed of transistors that are at roughly the same temperature. The reason is two-fold. First, the BB applied can be more accurately tuned to the needs of the transistors in the cell. Second, it is more likely that the circuit that we choose to calibrate the BB (Section 3.2.2) to apply to the rest of the cell be representative of the transistors in the cell. On the other hand, a cell that includes both hot and cold transistors is unlikely to be effectively body-biased.

It is well known that the spatial distribution of the temperature in a chip under load follows the breakdown into architectural modules. For example, the execution unit is hot whereas the L2 cache is cold. Consequently, we propose organizing the chip into BB cells that follow the contours of groups of hot and groups of cold architectural modules.

Critical Paths in Logic versus Memory

Different architectural modules have different types of critical paths. This is most obvious by comparing logic blocks such as functional units to memory structures such as the L1 cache or TLB. In the former, a critical path contains little wire and many, physically close gates; in the latter, it has a lot of wire and few, physically separated gates. Specifically, in logic blocks, the critical path in high-end processors is likely to be 8-16 FO4-equivalent gates [24, 27] connected by short wires. In contrast, in memory structures such as the L1, the critical path is likely to be the circuit that extends from a driver that drives a word line, through the word line, a pass transistor, the bit

line, and then to the logic at the sense amplifier. This path is likely to determine the cycle time in future high-frequency memory structures because it is hard to pipeline [13]. It is composed of (approximately) three logic gates connected by long wires.

From a V_{th} variation point of view, these two critical paths differ dramatically. The transistors in a logic path are many and physically close. Their large number enables a better averaging of random V_{th} variation effects. Also, physical proximity makes them subject to the same systematic V_{th} variation effect. On the other hand, the transistors in the memory path are few and distant from each other. For example, for the L1 used in our experiments, the (manhattan) distance between the gates at the two ends of the critical path is 0.55mm. Fewer transistors mean less averaging of random V_{th} variations; farther distances imply better averaging of systematic V_{th} variations. In addition, wires may also be affected by process variation; in this first study, however, we do not consider wire variation.

From this discussion, it follows that a given BB application changes the frequency-leakage operating point of a logic and a memory module differently. Consequently, given a cell with various architectural modules at the same temperature, we partition it into two cells, one with the logic and the other with the memory modules. For a similar reason, we also separate large memory modules from small ones (Section 3.5.4).

Additional Considerations

There are additional considerations related to AA-FGBB. First, when applying it to L2, we could decide not to use any forward BB in L2 — only use reverse BB wherever needed. The rationale is that L2 paths are unlikely to be critical; they are likely to have enough timing slack to absorb V_{th} variation effects. Avoiding forward BB limits leakage power. Since we want our experiments to be conservative, we do not use this optimization.

Second, we still break down large architectural modules such as the L2 into multiple BB cells (of rectangular shape). The reason is that a large module covers areas with wide-ranging V_{th} values.

Finally, we set the cell size between a maximum and a minimum value. The maximum size is given by the ϕ parameter of our V_{th} model (Section 3.3.1). If the cell is much larger than a circle of diameter ϕ, it is likely to contain multiple V_{th} "correlation regions" and, therefore, work

suboptimally with a single BB value. In addition, it will be hard to find a representative critical path for BB calibration.

The minimum size is given by the area overhead of setting up a cell. Such overhead is small, as discussed in Section 3.2.2. In our evaluation (Section 3.5), the chip with the most BB cells has cells of, on average, 1mm^2 at 45nm. This area is large enough for us to neglect BB area overhead.

3.4.2 How to Apply AA-FGBB

We envision the chip manufacturer taking a batch of recently-fabricated chips and applying AA-FGBB to them individually. His goal is to bring each chip to the best possible frequency-leakage operating point before shipment. This process will fix the body biases of each chip, and is performed after the chip is brought to appropriate temperature conditions. We suggest using a (conservative) temperature map generated after running representative workloads. It may be too conservative to bring the chip to a uniform, very high temperature (110 °C) as Tschanz *et al.* [81] do before applying BB. The reason is that the overestimated leakage reduces the opportunity to apply forward BB to increase frequency.

The chip designers have divided the chip into architecture-aware cells. In each cell, they have inserted a few, uniformly-spaced critical path samples that will be used to calibrate the BB to apply to the cell.

To apply AA-FGBB to a chip, we initially set a *Target Frequency* ($\mathcal{F}$) for it. It can be a fixed percentage of the *Nominal Frequency* (F_0) of the batch, namely the frequency that would be supported if there were no V_{th} variation. We feed $\mathcal{F}$ to the BB-calibrating circuitry in each cell of the chip. The circuitry automatically times the few critical-path samples in the cell and, based on the slowest one, sets the BB for the cell.

After all the cells in the chip have been body-biased, we measure the chip's new frequency and total power. If the latter is below a tolerable maximum value, we increase $\mathcal{F}$; otherwise we decrease it. In either case, we loop back to recompute the BB for the cells. We repeat this loop multiple times, decreasing $\Delta\mathcal{F}$ until we find the highest possible frequency subject to the power constraint.

3.5 Evaluation Methodology

We model batches of 200 processor chips, where each chip has its own personalized V_{th} map and critical paths. The T map is common to all chips (although T is not constant).

3.5.1 Processor Chip Model

We model a high-performance processor chip at 45nm. For this, we took the Alpha 21364 chip floor-plan, removed the router and I/O pads and added a large L2 cache as in [70]. Then, we scaled it to 45nm using CACTI [69] and data from [42]. Finally, we put four such units on a chip, and interconnected them with a wide snoopy bus. The resulting 132 mm^2 chip is shown in Figure 3.9(a). Each processor has a 64KB L1 I-cache, a 64KB L1 D-cache, and a private 2MB L2 cache.

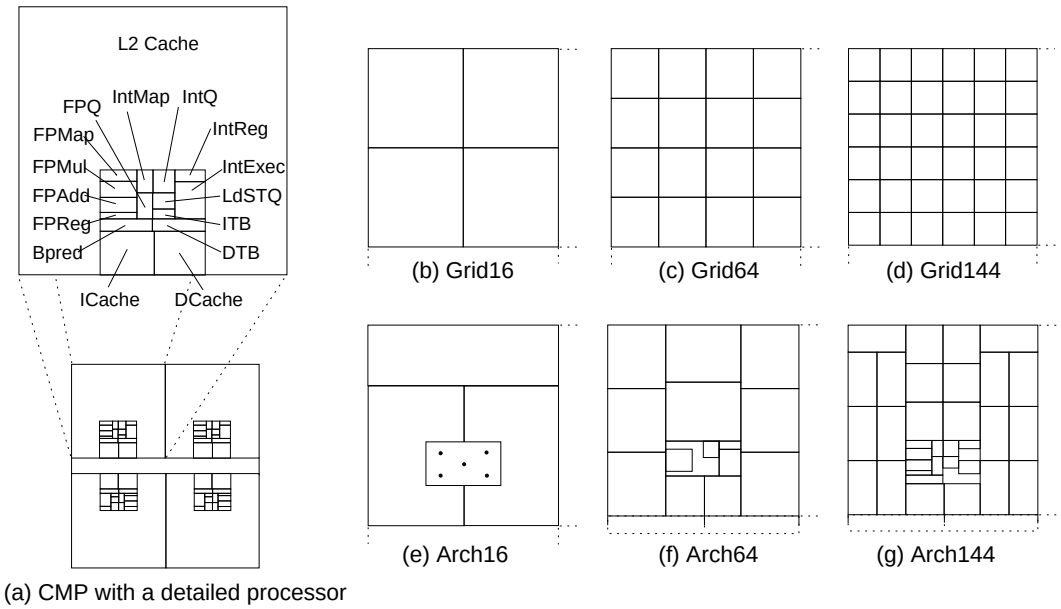

Figure 3.9: Multi-processor chip floor-plan used (a) and the partitioning of *one processor* and its share of the bus into regular BB cells (b–d) or architecture-aware ones (e–g). Chart (e) shows the five critical-path samples for one cell.

3.5.2 V_{th} Variation Model

Table 3.1 shows some of the process parameters used, including those for our V_{th} model. We target 6GHz, which we believe is realistic for 45nm. Unfortunately, there is no public-domain information that can suggest the likely values for V_{th} σ/μ and ϕ. For σ/μ, the 1999 ITRS [33] gave a design target of 0.06 for year 2005 (although no solution existed); however, the projection has been discontinued since 1999. On the other hand, Kahng [37, 38] reckons that the ITRS variability projections (for at least the gate-length parameter that he examines) are too optimistic. Rather than the $\sigma/\mu = 0.03$ for gate length reported by the ITRS, he wonders about possible 0.05 or 0.17 values. Consequently, in our experiments, we examine a V_{th} σ/μ ranging from 0.03 to 0.12, and report the main results with 0.12 and 0.09.

Parameter Values
Technology: 45nm; Frequency: 6GHz; V_{DD}: 1V V_{th}: μ: 150mV at 100 °C σ/μ: 0.03, 0.06, 0.09, 0.12 ϕ (fraction of chip's width): 0.1, 0.2, 0.5
Number of grid cells per chip: 16, 64, or 144 Number of architectural cells per chip: 16, 64, or 144 Number of samples per cell: 5 Number of chips per batch: 200

Table 3.1: Process parameters used in the evaluation.

For ϕ, we examine values ranging from 10% of the chip's width (short range of correlation) to 50% (long range). Since Friedberg *et al.*'s experiments [20] found that the gate-length parameter had a range close to 0.5, we report the main results with ϕ=0.5.

To generate the personalized V_{th} map of a chip with a given σ and ϕ, we use the geoR statistical package [64] of R [62]. We use a resolution of 1M points per chip. Each of our experiments uses a batch of 200 chips that have a different V_{th} map with the same σ and ϕ.

3.5.3 T Variation Model

We use HotSpot 3.02 [70] to estimate temperatures. HotSpot uses the combined dynamic and static energy dissipated. To estimate the dynamic energy, we use a cycle-accurate simulator of the chip

architecture augmented with the dynamic energy models from Wattch [10] and Orion [82]. These models are scaled to 45nm using technology projections from ITRS [34] except for frequency, since we target 6GHz. To estimate the leakage energy, we use HotLeakage 1.0 [85], also scaled using ITRS projections. The resulting leakage power is 19–39% of the total power, depending upon the application.

To estimate the on-chip T map, we use the iterative approach of Su *et al.* [75]: the temperature is estimated based on the current total power, the leakage power is estimated based on the current temperature, and the leakage power is added to the total power. This is repeated until convergence. Note that HotSpot also handles lateral heat conduction. In this way, we obtain the T map for one workload. We repeat this process for all workloads, creating multiple T maps T_i. Then, we conservatively build the final temperature map T_{final} that we use for the chips by setting, for each chip coordinate (x, y), $T_{final}(x, y) = \max_i\{T_i(x, y)\}$.

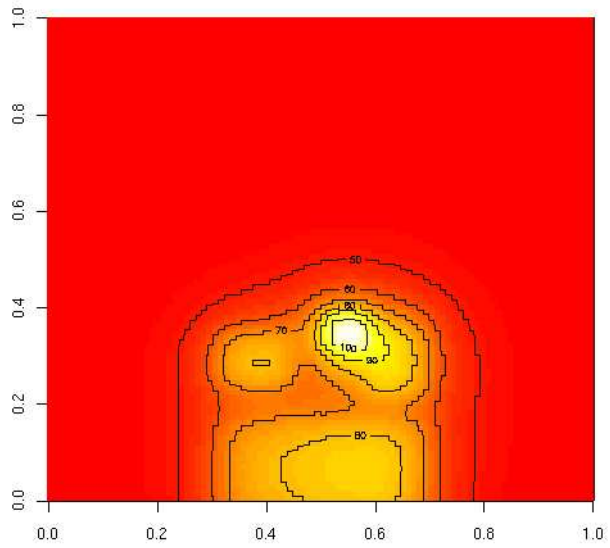

Figure 3.10: Temperature distribution of $T_{final}(x, y)$ (^{o}C).

The applications executed are nine SPECint and ten SPECfp applications. A workload consists of running four instances of the same application at a time, one on each core. We use the Ref input set and run the codes for 1 billion instructions after discarding the first 1 billion instructions. Looking at the resulting T map (Figure 3.10), we see that the maximum temperature reached in the chip is 109 ^{o}C. The hottest module is always the integer queue. Other hot modules include the

integer execution unit, load/store queue, integer register file and, for SPECfp, the FP execution unit.

3.5.4 Critical Path Model

Bowman *et al.* [8] estimate that a high-performance processor at this technology has about 10,000 critical paths. We distribute these paths uniformly across the chip area except for the L2s — we consider that the L2s do not have critical paths. As per Section 3.4.1, we build the critical paths differently in different modules. In the logic modules (e.g., the execution unit), the critical path delay is the delay of 12 close-by FO4 gates [24, 27]. For the memory modules, we use CACTI [69] to determine the optimal memory subarray sizes, the critical paths, and their physical layout. We distinguish two groups of structures: large memories (the two L1 caches) and small SRAM structures (e.g., the register file). The latter are assumed to cycle at twice the frequency of the former. The optimal subarray sizes are different in the two groups. In both cases, the critical path stretches from a driver that drives a word line, through the word line, a pass transistor, the bit line, and then to the logic at the sense amplifier. The critical path delay is the delay of three logic gates in the layout described, plus the wire delay corresponding to half the width of the subarray (word line) and the length of it (bit line).

The frequency supported by a chip is determined by the slowest of its 10,000 critical paths.

3.5.5 BB Environments Evaluated

We evaluate the environments of Table 3.2. We compare chips with $V_{\rm th}$ and T variation but no BB (*NoBB*) to the same chips with FGBB. FGBB chips can be architecture aware (AA-FGBB) or non-architecture aware (NAA-FGBB). The former are labeled *Arch*, while the latter are *Grid*. Their name is extended with a suffix that indicates the number of BB cells that the *full chip* contains.

The *Grid* approach divides the chip into a grid of 16, 64, or 144 cells (one processor is shown in Figures 3.9(b) to (d)). We choose rectangular cells for NAA-FGBB to follow the state-of-the-art, as set by Tschanz *et al.*'s cells [81].

The *Arch* approach uses the same number of cells. However, it first separates groups of hot modules from groups of cold ones. Then, in each of these groups, it separates logic, large memories,

Name	V_{th} Varies?	BB?	FGBB?	Arch Aware?	# of BB Cells
NoVar	N	–	–	–	–
NoBB	Y	N	–	–	–
CoarseBB	Y	Y	N	N	1
Grid16	Y	Y	Y	N	16
Grid64	Y	Y	Y	N	64
Grid144	Y	Y	Y	N	144
Arch16{T, T&CP}	Y	Y	Y	Y	16
Arch64{T, T&CP}	Y	Y	Y	Y	64
Arch144{T, T&CP}	Y	Y	Y	Y	144

Table 3.2: Chip environments evaluated.

and small memories (Figures 3.9(e) to (g)). For each number of cells, we have two schemes, *ArchT&CP* and *ArchT*. Our proposal is *ArchT&CP*, and considers both temperature and type of critical path variations. *ArchT* only considers temperature — since it does not care about the module type, all modules are sampled with logic-type critical paths (12 FO4 gates), which reduces BB effectiveness. We use this approach to assess the two components of AA-FGBB so that we can keep the same cell shapes for both *ArchT&CP* and *ArchT*. Changing the cell shapes would cloud the comparison.

As a reference, we also consider chips with a single BB cell (*CoarseBB*) and with no process-induced V_{th} variation (*NoVar*). Note that *NoVar*'s V_{th} is not constant due to T variations.

Each cell has five uniformly-spaced critical path samples (Figure 3.9(e) shows them for one cell). To set the BB, we estimate the cell frequency from the slowest of such samples. Although the cells in the L2s have no critical paths, they still have critical-path samples copied from L1 cells. We use them to determine the BB for L2 cells.

3.6 Evaluation

We first assess the effect of V_{th} variation. Then, we evaluate AA-FGBB either targeting frequency increase or leakage reduction. Unless otherwise noted, all experiments use the T map of Section 3.5.3.

3.6.1 Characterizing V_{th} Variation

Figure 3.11 uses our V_{th} model to show chip frequency and chip leakage power as V_{th} variation (measured in σ/μ) changes. For each value of σ/μ, the figure shows bars for three different ϕ. In all cases, frequency and leakage are relative to the *NoVar* chip. The bands in the bars show the variation across chips in the batch.

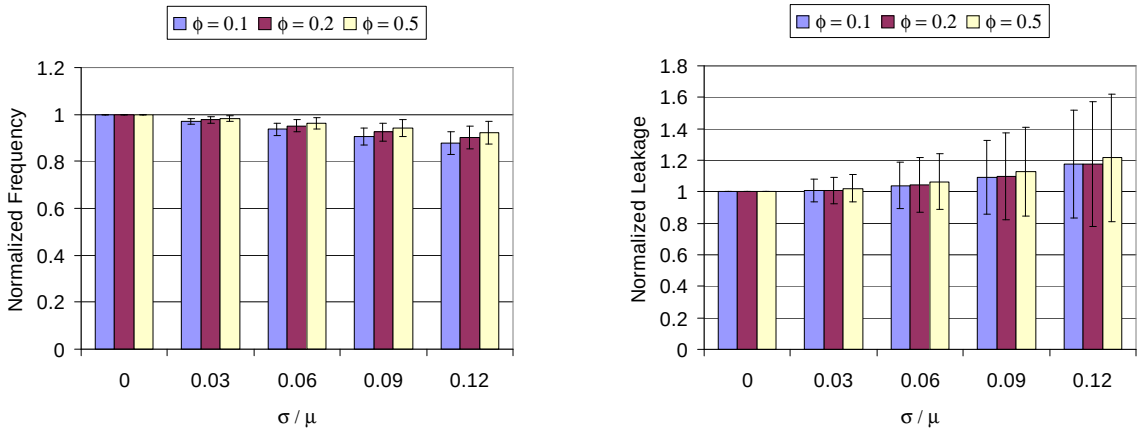

Figure 3.11: Impact of V_{th} variation on the chip's frequency and leakage power.

As V_{th} variation increases, the average frequency of the chip decreases and its average leakage power goes up quickly. On average, at 0.12 variation, the frequency is 10% lower and the leakage 20% higher. Clearly, variation is undesirable.

The long bands show high variation across chips in the batch. This is due to the T variation. At high T, a transistor becomes slower and leakier. Consequently, if transistors with very high V_{th} happen to be in the hottest region of the chip, the chip is likely to have low frequency. On the other hand, if many transistors with very low V_{th} are in the hottest area, the chip is likely to have high leakage.

We see two main trends. First, variation across chips is higher in leakage power — since leakage is exponential with T, an unfavorable V_{th} distribution can be really bad. Second, the average frequency is lower for low ϕ. The reason is that, given a set of high-V_{th} transistors, if they are uniformly spread out in the chip (low ϕ), there is a higher chance that some will be in the hottest region of the chip, thus sinking the chip's frequency. Therefore, with low ϕ, more chips will be slow.

We have repeated the experiments with the whole chip at constant T. In this case, there are neither variation bands nor differences between different ϕ.

3.6.2 Impact of AA-FGBB on Frequency Binning

The most fruitful use of FGBB is to tune the chips in a batch so that they fall into desirable frequency-leakage bins [6]. The goal is to place each chip at the highest possible frequency bin where it still meets the power consumption constraint.

Acceptable Region

According to Tschanz *et al.* [81], the *Acceptable Region* for a chip is bounded by two conditions: (i) the frequency should be higher than a given minimum value, and (ii) the sum of dynamic and leakage power should be less than a given maximum value. These constraints are represented by a horizontal and a slanted line, respectively, in a frequency-leakage plot. For our batch of 200 chips, these lines are shown in Figure 3.12(a). The figure shows a scatter plot of the frequency and leakage power of the chips, with the axes normalized to the *NoVar* environment (no process-induced V_{th} variation). The slanted line has this shape because, as frequency increases, the dynamic power increases linearly and, therefore, the amount of tolerable leakage power decreases linearly. Inside the Acceptable Region, higher frequency is better.

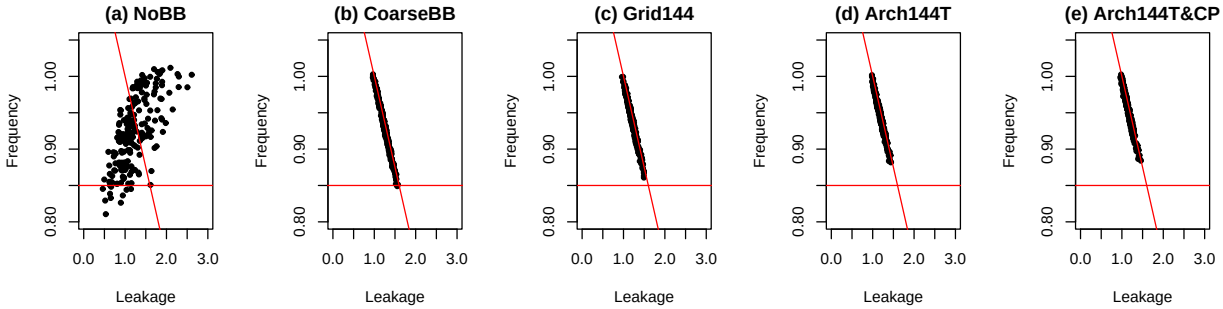

Figure 3.12: Frequency versus leakage power for a batch of chips under various schemes. The experiments correspond to frequency binning for $\phi = 0.5$ and $\sigma/\mu = 0.12$.

Appendix B.3 justifies how we set the lines that bound the Acceptable Region in Figure 3.12(a). In particular, the low frequency line is set to 0.85 times the frequency of the *NoVar* chip. This

gives an approximate acceptable frequency range of 15%, which is similar to the 3.0–3.6 GHz range of Intel's Xeon family at 90nm [31]. Following the Xeon family, in our experiments, we divide the area into four equally-spaced frequency bins. As a fraction of the *NoVar* frequency, the range of the bins is: 0.850–0.887, 0.887–0.925, 0.925–0.962, and over 0.962.

Processors in the top bins fetch a hefty price premium. For example, Xeon Nocona processors of 3.6, 3.4, and 3.2 GHz are priced about 3x, 2x, and 1.25x higher, respectively, than those of 3.0 GHz in November 2005 [84].

Chip Placement

Our goal is to move every chip to the Acceptable Region, at its maximum possible frequency. For each chip, we apply the algorithm described in Section 3.4.2, which gives us the highest frequency at which the chip still falls inside the Acceptable Region. Some chips leak too much even at the lowest frequency of the region and, therefore, are discarded.

An understanding of how FGBB changes the frequency-leakage conditions of a chip enables the algorithm of Section 3.4.2 to perform an efficient search for the optimal target frequency. Figure 3.13 takes the chip closest to the average frequency and leakage power in the batch of Figure 3.12(a) and applies either *Arch144T&CP* or *Grid144*. For both schemes, we vary $\mathcal{F}$ from 1.0 times *NoVar*'s frequency (where leakage power is high) to 0.85 times (where leakage power is low). At each step, we plot the frequency and the leakage measured.

The figure shows that FGBB is able to modify the operating conditions of the chip significantly. For example, *Arch144T&CP* can trade off a 14% frequency change for a 13x leakage power change. Most importantly, the figure shows that *Arch144T&CP* is able to move the chip to a *better region* than *Grid144* — namely, to a higher frequency for the same leakage. Consequently, *Arch144T&CP* should be able to place the processors in better bins. While the frequency boost may seem small in the figure, it may have a significant revenue impact to a large vendor.

Results Obtained

We take the original batch of chips (Figure 3.12(a)) and apply our binning algorithm using the *CoarseBB*, *Grid144*, *Arch144T*, or *Arch144T&CP* schemes. The resulting frequency-leakage scatter

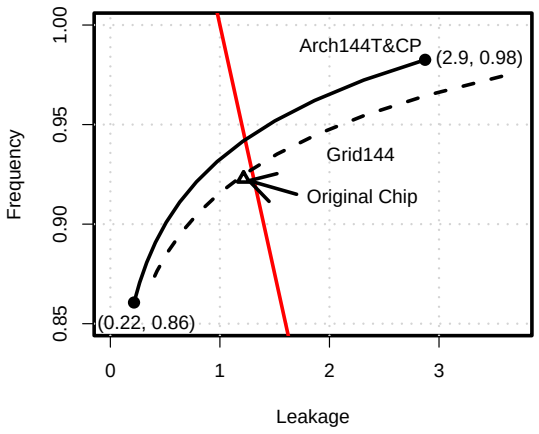

Figure 3.13: Changing the frequency and leakage power of a chip with *Arch144T&CP* or *Grid144*.

plots are shown in Figure 3.12(b)–(e). The charts show that all the schemes are able to move practically all the chips into the Acceptable Region, in the area close to the slanted line. However, the schemes differ in how high the chips have been pushed. The AA-FGBB schemes are more effective at reaching high frequencies.

The differences are best seen in Figure 3.14, which shows the fraction of the 200 chips that fall in each frequency bin for the different schemes. Chart (a) corresponds to our experiment, while (b) repeats it for $\sigma/\mu = 0.09$.

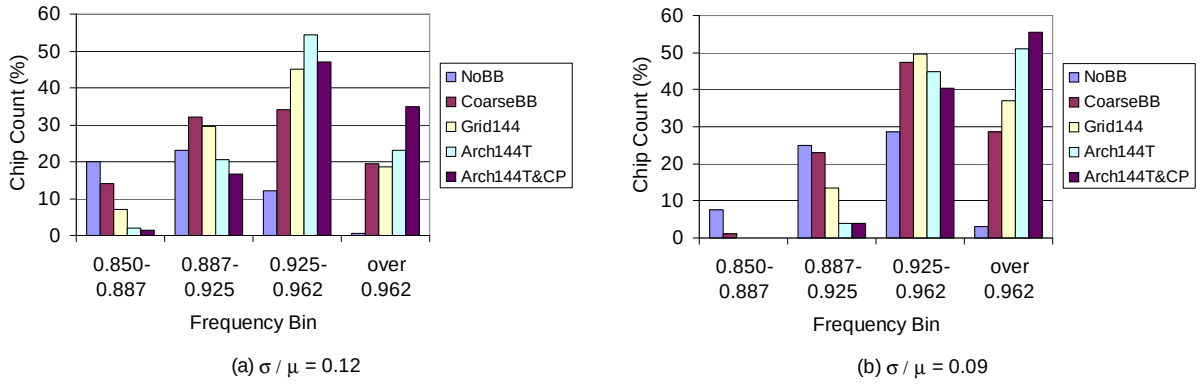

Figure 3.14: Frequency binning obtained by the different FGBB schemes, for $\phi = 0.5$ and either $\sigma/\mu = 0.12$ (a) or $\sigma/\mu = 0.09$ (b).

Chart (a) shows that the AA-FGBB schemes move many chips to the top bins. Specifically,

while *Grid144* only has 18% of the chips in the top bin, *Arch144T&CP* has 35% of them there. This *doubles* the number of such chips, and represents a major revenue increase for the manufacturer — chips in the top bin can be priced 50% higher than those in the next bin [84]. It also demonstrates this study's thesis that FGBB should be applied in an architecture-aware manner.

While *Arch144T* is more effective than *Grid144*, it is significantly inferior to our proposed *Arch144T&CP*. Consequently, both temperature and type of critical path awareness contribute to the gains.

Comparing *Arch144T&CP* to *NoBB*, we see the overall impact of AA-FGBB. While *NoBB* only has 14% of the chips in the top two bins, *Arch144T&CP* has 82% of them there. This is an even larger revenue differential, since chips in the top two bins can be priced 3x and 2x higher, respectively, than those in the bottom bin [84].

Chart (b) shows that the trends are the same for $\sigma/\mu = 0.09$. While the improvement as we go from *NoBB* to *Grid144* and then to *Arch144T&CP* is slightly smaller, it is still substantial. Specifically, the number of chips in the top bin still changes from 37% in *Grid144* to 56% in *Arch144T&CP*. Consequently, our results are valid for more conservative variation values.

For completeness, Figure 3.15 repeats the experiments for small ϕ. *Arch144T&CP* still greatly improves over *Grid144*.

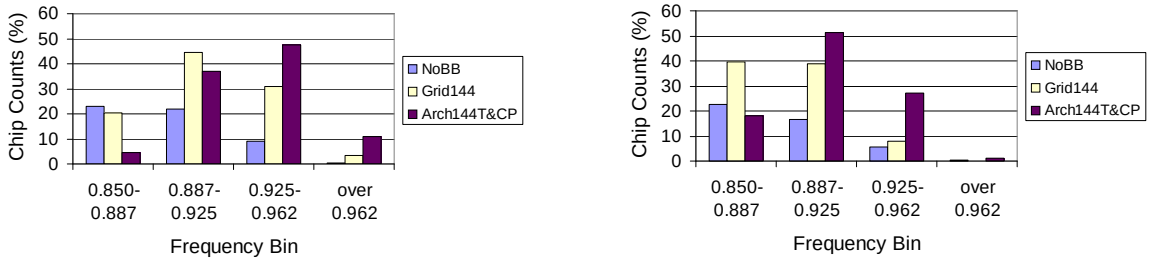

Figure 3.15: Frequency binning obtained for $\phi = 0.2$ (a) or $\phi = 0.1$ (b).

3.6.3 Impact of AA-FGBB on Leakage

Another use of FGBB is to try to reduce the leakage power of each chip in the batch without changing its frequency. To do so, we take each chip in turn, measure its frequency, set the target

frequency ($\mathcal{F}$) to it, and apply FGBB. In reality, the chip ends up having a slightly different frequency due to sampling inaccuracies and variation effects. Consequently, we may have to iterate a few times with slightly different $\mathcal{F}$.

Figure 3.16 shows how the original frequency-leakage scatter plot (*NoBB*) changes when we apply the algorithm using the *CoarseBB*, *Grid144*, *Arch144T*, or *Arch144T&CP* schemes. As usual, the axes are normalized to *NoVar*. Below each chart, we have the mean leakage power consumed by the chips in the batch.

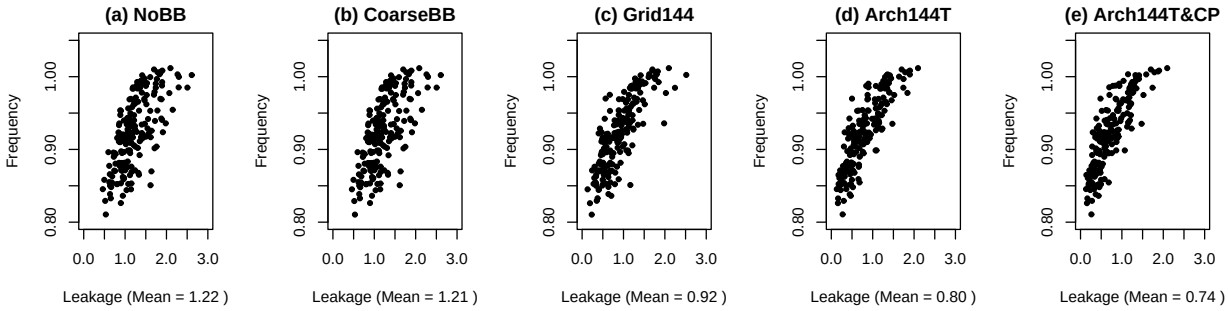

Figure 3.16: Frequency versus leakage power for a batch of chips under various schemes. The experiments correspond to reduced leakage for unchanged frequency, $\phi = 0.5$ and $\sigma/\mu = 0.12$.

Comparing Chart (a) to (c)–(e), we see that, if we apply FGBB, most of the chips move to the left noticeably. In all cases, while the frequencies of the chips do not change, the overall leakage decreases substantially. *Arch144T&CP* is very effective, reducing the mean leakage per chip by 40% over *NoBB*. *Arch144T* is also effective, as it manages a 34% leakage reduction over *NoBB*. *Grid144* is less so, as it only accomplishes a 25% reduction. This result indicates that FGBB should use architecture awareness.

Chart (b) shows that *CoarseBB* is unable to reduce the leakage power.

Figure 3.17 extends these experiments to all the BB environments, although we only show the mean leakage power of the chips. For 64 and 144 cells per chip, our proposed *ArchT&CP* scheme has the lowest leakage for a given number of cells. The *ArchT* scheme is slightly less effective, and the *Grid* scheme is even less so. Therefore, for best performance, we recommend AA-FGBB with both temperature and type of critical path awareness. When the number of cells is 16, *Grid* performs relatively well. This is due to a pathological critical-path case. In any case, as the number

of FGBB cells decreases, FGBB's effectiveness goes down. These particular chips need 64 cells or more.

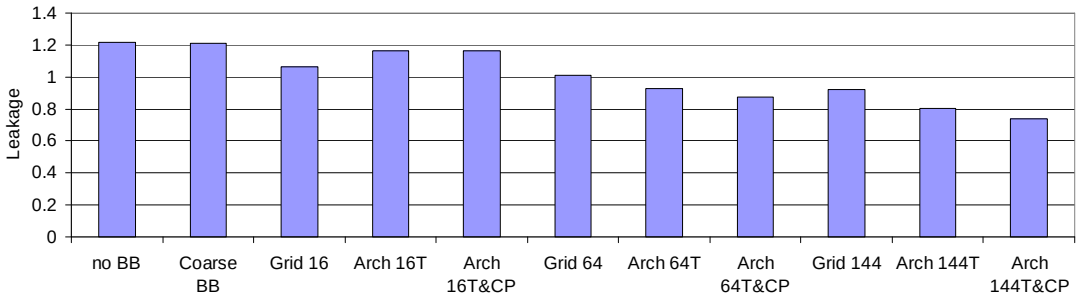

Figure 3.17: Mean leakage power of the chips in the batch for different BB schemes.

Chapter 4

Related Work

4.1 Work Related to ReViveI/O

Masubuchi *et al.* [48] proposed adding a PDD to the kernel to support disk I/O recovery. Their scheme corresponds to *Stall* in Section 2.5, which blocks disk writes until the next checkpoint [47]. ReViveI/O differs as follows: (i) instead of blocking, it buffers the data and commits them later in the background, and (ii) it supports network I/O. More importantly, this study contributed with the full implementation, testing, and evaluation of an efficient I/O undo/redo prototype compatible with solutions such as ReVive [61] or SafetyNet [72].

High-availability machines such as HP's Nonstop Architecture [25] and IBM's S/390 mainframes [71] attain fault tolerance through expensive hardware support, often involving extensive component replication. We seek a less expensive design point.

Sequoia [3] is an SMP where mirrored main memory is considered to be the checkpointed state. When a dirty line is evicted from a cache, a checkpoint is triggered. A checkpoint is also forced every time a processor requests I/O, which is an inefficient approach.

OS- and library-based checkpointing includes UNICOS [41], KeyKOS [44], diskless checkpointing [60], and fault-tolerant Mach [66]. Their checkpoint interval is typically minutes or longer. This results in high MTTR and would induce intolerably long message delays. Some schemes provide disk I/O recovery by journaling, but none addresses network I/O recovery.

Database management systems (DBMS) have their own well-known I/O recovery mechanisms [21]. Checkpoint intervals are in the order of minutes to achieve low enough overheads, and the recovery process takes tens of minutes [49] or more. For example, for System B of [49], targeting a 5-minute database recovery time, incurs 8.2% overhead in throughput. Note that such database recovery

time does not include the system recovery time (e.g., repair and reboot), which at least adds minutes. Similarly, targeting a 20-minute database recovery time incurs 6.6% overhead. Also, the fault results in the loss of ongoing transactions. In contrast, for the frequent faults (MR faults), ReVive plus ReViveI/O has three advantages: (i) ongoing transactions are not lost, (ii) the system recovers in less than 1 second, and (iii) applications and kernel need no modification. When a DBMS runs on ReVive with our I/O undo/redo layer, only the infrequent faults (NMR faults) trigger database recovery. Consequently, the DBMS can use lower-overhead checkpointing, at the expense of longer recovery time.

There are cluster options for databases such as Oracle RAC (Real Application Clusters) [57]. These solutions use multiple machines to provide higher system availability. However, to achieve good performance, they require major changes to the database management system and, possibly, to the applications that run on top of it.

Our mechanisms overlap with ideas from other works. Hu and Yang [28] proposed the Disk Caching Disk (DCD), mainly to boost the performance of random disk writes; our disk buffer area is like a DCD for reliability. Several systems, such as the Legato Prestoserve [51] and Baker *et al.*'s scheme [2] have used NVRAM to speed up disk writes; we use NVRAM to speed up writes within checkpoints. Our PDD can be thought of as a thin Virtual Machine (VM) for I/O; VMs have been used for forward error recovery [9] and for post-intrusion replay [16], rather than for rollback recovery of faults.

ReViveI/O is also related to transactional memory systems (TMS) [23]. TMS share similar I/O issues with hardware-based memory checkpointing systems: when a transaction is aborted, TMS need to undo any work done by that transaction, including I/O. On the other hand, there are some differences: (i) TMS do not need to redo I/O after a transaction is aborted, (ii) TMS have less emphasis on durability than memory checkpointing systems because their primary purpose is to support atomicity, not reliability, and (iii) TMS may additionally need some mechanism to detect and track the dependences and conflicts between threads that are induced by I/O operations.

4.2 Work Related to Architecture-Aware Fine-Grain Body Biasing

The problem of process variation has been known to the VLSI community for more than a decade, but Borkar [4] is one of the first to take a computer architecture perspective. Borkar *et al.*'s efforts at Intel to tackle this problem are summarized in [6, 5]. One of the approaches they propose is BB, which is summarized in Section 3.2.

Stine *et al.* [74], Orshansky *et al.* [58], and Friedberg *et al.* [20] performed actual measurements of test chips to characterize the magnitude and spatial distribution of dielectric thickness or gate length variation. They observed that a significant portion of the WID variation is systematic. Cao and Clark [11] proposed a model that attributes V_{th} variation to gate length variation and studied the impact of spatial correlation on the delay of one critical path. For our V_{th} model, we build on Friedberg *et al.*'s model. However, we have to additionally include the largely random effects of material doping.

Microarchitecture work on variability was done by Bowman *et al.* [8], who derived a formula for computing chip frequency by modeling critical paths in the chip in the presence WID process variation. We use their formula in our work. The model was later extended by Marculescu and Talpes [45] by assuming that the number of critical paths per pipeline stage is proportional to the stage's device count. They applied the model to GALS processors. Datta *et al.* [15] developed a statistical approach for pipeline delay analysis to show the importance of logic depth in variability analysis. Finally, Nassif [55] claimed that, in addition to PVT variations, wire delay variation is important.

Chang and Sapatnekar [12] and Srivastava *et al.* [73] considered the impact of spatially correlated variation on leakage and/or performance. They generated distributions on coarse grid points by assuming a linear [12] or reciprocal [73] correlation function. They did not discuss trade-offs between leakage/frequency and the various correlation lengths.

Although no technical detail is publicly available, it is claimed that Transmeta's LongRun2 [80] is able to dynamically adjust V_{th} to control leakage power. This scheme is thought to be effective for D2D V_{th} variation.

Chapter 5

Conclusions

5.1 ReViveI/O

The main contribution of this work is the full implementation, testing, and experimental evaluation of *ReViveI/O*, an efficient scheme for I/O undo/redo that is compatible with high-frequency checkpointing architectures such as ReVive and SafetyNet. In addition, we perform a sensitivity analysis of what checkpoint frequencies are required to maintain acceptable throughput and response times. Overall, this work completes the viability assessment of such novel memory-recovery architectures.

Our ReViveI/O prototype shows that low-overhead, tiny-MTTR recovery of I/O is feasible. For 20–120 ms between checkpoints, the throughput of a throughput-oriented workload such as TPC-C on Oracle decreases by no more than 1%. Moreover, for 50 ms between checkpoints or less, the response time of a latency-bound workload such as WebStone on Apache remains tolerable. In all cases, the recovery time is practically negligible. Moreover, kernel, device drivers, and applications remain unmodified. Finally, the combination of ReVive and ReViveI/O is likely to reduce the throughput of TPC-C-class applications by 7% or less for 60–120 ms checkpoint intervals, while incurring a tiny MTTR of less than 1 second.

Our analysis suggests that support for ReVive plus ReViveI/O makes for a very cost-effective high-availability server. In general, compared to software-based recovery solutions (OS, library, and database), our approach has a higher error coverage, lower overhead, and much smaller MTTR. We estimate that it delivers a 2–3 orders of magnitude reduction in unavailability over a database server that we evaluated. Finally, compared to hardware-intensive solutions such as HP NonStop systems, it is much cheaper while maintaining high availability.

An avenue for future work is to apply our techniques to transactional memory systems. We

expect that ReViveI/O can be applied to transactional memory systems by adding support for per-thread output commit and for some dependence/conflict tracking mechanism through the Memory Buffer.

5.2 Architecture-Aware Fine Grain Body Biasing

Parameter variation is the next big challenge for processor designers as technology moves below 65nm. While technology to handle variability will surely be developed at the lithographic, circuit and layout levels, we believe that computer architecture techniques will play a key role in coping with parameter variations. This study has introduced and demonstrated one such novel technique.

The main contribution of this work is to show that Fine-Grain Body Biasing (FGBB) is substantially more effective if cells are selected in an architecture-aware manner — largely following the shapes of architectural modules. The rationale for Architecture-Aware FGBB (AA-FGBB) is that architectural functionality is the main determinant of the temperature and the type of critical path in a section of the chip, which in turn greatly impact the optimal BB to apply. The second contribution is the development of a novel, parameterized model of WID V_{th} variation.

Our experiments with 200-chip batches show that AA-FGBB (*Arch144T&CP*) boosts chip frequency substantially. Specifically, it pushes 35% of the chips into the top frequency bin, compared to the 18% managed by advanced conventional FGBB (*Grid144*). Each of these improved chips can potentially be priced 50% higher. Moreover, it also moves 82% of the chips into the top two frequency bins, compared to the 14% that make it without any BB. These chips can potentially be priced 3x and 2x higher than those in the bottom bin. Finally, if *Arch144T&CP* targets leakage power, it reduces the average leakage of the chips by 40%, compared to the 25% reduction obtained with *Grid144*.

Our results strongly recommend the use of AA-FGBB with both temperature and type of critical path awareness. Moreover, the effectiveness holds for a wide range of V_{th} variation parameters.

Our current work involves providing an architecture-aware solution to voltage variation. We are also examining the use of module replication to address variability.

Appendix A

Detailed View of ReViveI/O Operations

In this appendix, we present a closer look at how ReViveI/O (*BufferVolatile* scheme) works. For simplicity, we consider only MR (memory-recoverable) faults and we assume that the fault detection latency is negligibly small. Also, without loss of generality, we consider only one output request and its performance. Note that the output is first *requested* by an application or by the operating system and later, it is *performed* to physical devices, at which point the output becomes *visible* to the outside world (the entities outside the scope of checkpoint and rollback recovery) for the first time.

We begin with defining all the events that are relevant to a single output.

Events	Definitions
`R`	The output is requested.
`P[`	The system starts performing the output.
`]P`	The system ends performing the output.
`C[`	Start of checkpointing.
`]C{`x`}`	End of checkpointing. (The checkpoint contains information about x, where x can be output request `R` or $\emptyset$.)

Table A.1: Definition of events related to a single output in ReViveI/O.

Figure A.1 shows a typical fault-free sequence of events involved with an output. It starts with the previous checkpoint (`]C{}`). Then an output is requested (`R`). The request is not immediately performed but buffered in PDD's data structure (*Memory Buffer*). Later the system takes a checkpoint, which includes the information about `R` held in *Memory Buffer*. After the checkpoint, the actual output is performed.

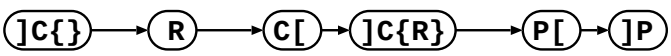

Figure A.1: An output in ReViveI/O without faults.

Figure A.2 augments Figure A.1 with all the possible MR faults. It also considers the case of long-lasting output operations that extend over multiple checkpoint intervals. The shaded region corresponds to the events that are visible to the outside world. The diagram starts with checkpoint i.

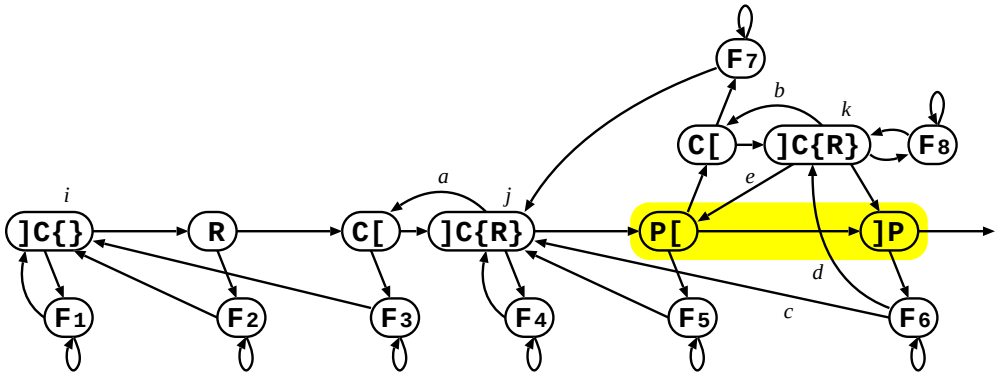

Figure A.2: An output in ReViveI/O with faults.

We look at each fault in turn.

- $\texttt{F}_1$: This is trivial. A fault happens after checkpoint i. The system rolls back to checkpoint i and resumes execution.

- $\texttt{F}_2$: When the system rolls back to checkpoint i, any information about `R` is lost. However, since this sequence of operations is hidden to the outside world, it does not incur any inconsistency.

- $\texttt{F}_3$: This is a fault while a checkpoint is being taken. It is handled in the same way as $\texttt{F}_2$.

- $\texttt{F}_4$: The system rolls back to checkpoint j, which contains information about `R`. Therefore, `R` is guaranteed to be performed later. (Transition a corresponds to the case where the system is busy and `R` waits long to be performed.)

- $\texttt{F}_5$: The system rolls back to checkpoint j and performs the output from the beginning. Before the rollback, the outside world might be able to see the partial output. If it is a disk write, the complete data will be written shortly (Here, we depend on the idempotent nature of disk writes). If it is a network data transmission, the partial packet is almost surely dropped

at the receiving end due to checksum error or length mismatch, and the correct packet will be resent later.

- $\mathtt{F}_6$: If no checkpoint has been taken after checkpoint j, the system rolls back to j (transition c) and R is performed from the beginning. On the other hand, if the output operation has taken so long that its duration involves one or more checkpoints (k), the system rolls back to the latest checkpoint of those (transition d). Note that since `P` was not finished when the system started taking checkpoint k, the information about `R` is not purged from *Memory Buffer* and is held in checkpoint k. Therefore, `R` will be performed from the beginning via transition e.

  In both cases, the output is performed more than once. We again depend on the idempotent nature of the output in consideration (i.e., disk writes and TCP data transmissions).

- $\mathtt{F}_7$: The system rolls back to checkpoint j and the output is performed from the beginning.

- $\mathtt{F}_8$: The system rolls back to checkpoint k and the output is performed from the beginning via transition e.

Note that although we draw self loops for the faults in Figure A.2 for the sake of completeness, simultaneous multiple faults or faults during recovery are not always memory-recoverable.

Appendix B

Mathematical Background of the Process Variation

B.1 Statistics

Correlation Function. The *covariance* of two random variables X and Y is defined as $cov(X,Y) = \mathrm{E}[(X-\mathrm{E}[X])(Y-\mathrm{E}[Y])]$. For a location-dependent random variable $Z(\vec{x})$, its correlation function is defined as $\rho(\vec{x},\vec{y}) = cov(Z(\vec{x}),Z(\vec{y}))/var(Z)$ ($|\rho| \leq 1$). In the spherical model, $\rho(\vec{x},\vec{y})$ is a function of $r \equiv |\vec{x}-\vec{y}|$ and is defined as:

$$\rho_\phi(r) = \begin{cases} 1-(3r/2\phi)+(r/\phi)^3/2 & \text{if } r \leq \phi \\ 0 & \text{otherwise.} \end{cases}$$

Note that since we are only considering WID variation, $\rho(\infty)$ is zero.

Leakage Computation. Under a constant temperature, a transistor's subthreshold leakage current is written as follows:

$$I_{\text{leak}} = I_0 e^{-qV_{\text{th}}/kT} \tag{B.1}$$

Given a normal distribution $N(\mu,\sigma^2)$ for V_{th}, the leakage current in the chip is the integral of Eq. (B.1):

$$I_{\text{leak}} = \int_{-\infty}^{\infty} \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sqrt{2\pi}\sigma} I_0 e^{-qx/kT} dx = I^0_{\text{leak}} e^{(q\sigma/kT)^2/2},$$

where $I^0_{\text{leak}} = I_0 e^{-q\mu/kT}$.

Frequency Computation. For a random variable Z, we denote the ratio of its standard deviation and its average by R_Z (i.e., $R_Z = \sqrt{var(Z)}/\mathrm{E}[Z]$). Under a constant temperature, the delay of a

gate can be written as:

$$T_g = \frac{KV}{(V - V_{\text{th}})^{\alpha}}, \tag{B.2}$$

where K is a technology-dependent constant and α is typically 1.3 [67]. Given a normal distribution $N(\mu, \sigma^2)$ for V_{th}, we write $V_{\text{th}} = \mu + \sigma\delta$, and δ is distributed $N(0, 1^2)$. For the common case when $|\delta| \ll (V - \mu)/\sigma$, we re-write Eq. (B.2) as $T_g \approx \frac{KV}{(V-\mu)^{\alpha}} \left[1 + \frac{\alpha R}{V/\mu - 1}\delta\right]$, where $R = \sigma/\mu$. This shows that the gate delay also follows a normal distribution $N(\mu_{Tg}, \sigma^2_{Tg})$, where $\mu_{T_g} = \frac{KV}{(V-\mu)^{\alpha}}$ and $\sigma_{T_g} = R_{T_g} \cdot \mu_{T_g} = \frac{\alpha R}{V/\mu - 1}\mu_{T_g}$.

Let T_{cp} be the delay of a critical path. The average value of T_{cp} is $\mu_{T\text{cp}} = n_{\text{cp}}\mu_{Tg}$, where n_{cp} is the number of gates in the critical path. Let $T_{gi} = \mu_{T_g} + \sigma_{T_g}\delta_i$ $(1 \le i \le n_{\text{cp}})$ be the delay of the ith gate in the critical path (δ_i follows $N(0, 1^2)$). Then, the variance of T_{cp} is computed as:

$$\begin{aligned}
\sigma^2_{T\text{cp}} &= \text{E}[\{\sum_{i=1}^{n_{\text{cp}}}(T_{gi} - \mu_{T_g})\}^2] = \sigma^2_{T_g}\left(\sum_{i=1}^{n_{\text{cp}}}\text{E}[\delta_i^2] + \sum_{i\neq j}\text{E}[\delta_i\delta_j]\right) \\
&= \sigma^2_{T_g}(n_{\text{cp}} + \sum_{i\neq j}\rho(d_{ij})),
\end{aligned}$$

where ρ is the correlation function of V_{th} and d_{ij} denotes the distance between the ith and jth gates i in the critical path. We assumed that the correlation function is uniform and isotropic. In the spherical model, $\sigma_{T\text{cp}}$ depends on the distance between the gates in the path relative to ϕ. If the distance between the gates is $\ll \phi$, their V_{th} is very correlated ($\rho(d_{ij}) \approx 1$). If we assume perfect correlation, the critical path delay is normally distributed, the variance is highest and $\sigma_{T\text{cp}}/\mu_{T\text{cp}} = \sigma_{Tg}/\mu_{Tg}$. If, instead, the distance between the gates is high compared to ϕ, the gates are more independent of each other ($\rho(d_{ij}) \approx 0$). In the limit, if we assume no correlation, the critical path delay is also normally distributed, the variance is lowest because the random component tends to cancel itself and $\sigma_{T\text{cp}}/\mu_{T\text{cp}} = \sigma_{Tg}/\sqrt{n_{\text{cp}}}\mu_{Tg}$.

We derive the probability distribution of the longest delay, $\max\{T_{\text{cp}}\}$, and the chip frequency. For a random variable Z, let $\text{PDF}_Z(x)$ and $\text{CDF}_Z(x)$ denote the probability density function and the cumulative density function, respectively, of Z. Let us assume that T_{cp} follows $N(\mu_{T\text{cp}}, \sigma^2_{T\text{cp}})$.

Following the argument of Bowman *et al.* [8],

$$\mathrm{CDF}_{\max\{T_{\mathrm{cp}}\}}(x) = \Pr(\max\{T_{\mathrm{cp}}\} \le x) = \mathrm{CDF}_{T_{\mathrm{cp}}}(x)^{N_{\mathrm{cp}}},$$

where N_{cp} is the number of independent critical paths in a chip. Therefore,

$$\mathrm{PDF}_{\max\{T_{\mathrm{cp}}\}}(x) = \frac{d}{dx}\mathrm{CDF}_{\max\{T_{\mathrm{cp}}\}}(x) = N_{\mathrm{cp}}\mathrm{PDF}_{T_{\mathrm{cp}}}(x)\mathrm{CDF}_{T_{\mathrm{cp}}}(x)^{N_{\mathrm{cp}}-1}.$$

Since the chip frequency f is inversely proportional to $\max\{T_{\mathrm{cp}}\}$, $\mathrm{PDF}_f(x) = x^{-2}\mathrm{PDF}_{\max\{T_{\mathrm{cp}}\}}(x^{-1})$. If we consider the normalized frequency f/f_0 ($f_0 = \mu_{T_{\mathrm{cp}}}^{-1}$), we have

$$\mathrm{PDF}_{f/f_0}(x) = x^{-2}\frac{N_{\mathrm{cp}}}{\sqrt{2\pi}R_{T_{\mathrm{cp}}}}\exp\left[-\frac{(x^{-1}-1)^2}{2R_{T_{\mathrm{cp}}}^2}\right]\left[\frac{1}{2}\left(1+\mathrm{erf}\left(\frac{x^{-1}-1}{\sqrt{2}R_{T_{\mathrm{cp}}}}\right)\right)\right]^{N_{\mathrm{cp}}-1},$$

where erf is the error function and $R_{T_{\mathrm{cp}}} = \sigma_{T_{\mathrm{cp}}}/\mu_{T_{\mathrm{cp}}}$. When the gates in a critical path are fully correlated, T_{cp} has the largest variation and $R_{T_{\mathrm{cp}}} = R_{T_g} = \frac{\alpha R}{V/\mu - 1}$. Figure B.1 shows the distribution of chip frequency for such case for various values of N_{cp}.

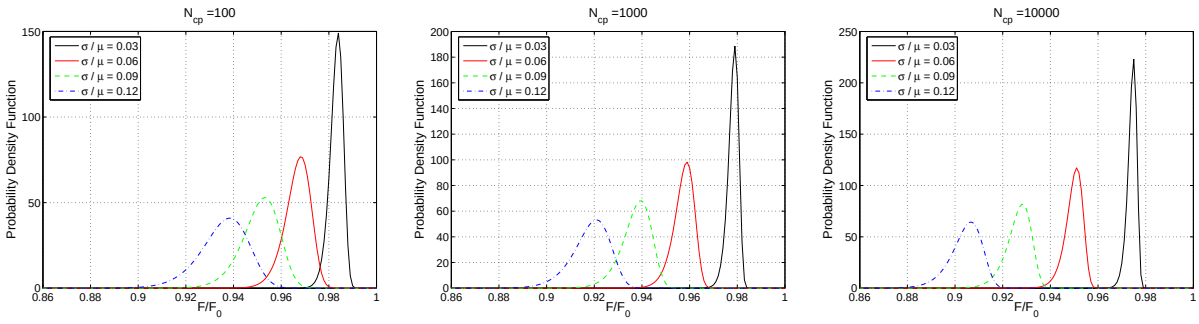

Figure B.1: Probability distribution of chip frequency for $N_{\mathrm{cp}} = 100$, 1000, and 10000.

B.2 Theory of Die-to-Die and Within-Die Variations

Die-to-die (D2D) variation can be modeled as V_{th} of all the gates in a die having a same offset from the average V_{th} of a wafer: $V_{\mathrm{th}}^{\mathrm{D2D}} = V_{\mathrm{th}} + \Delta V_{\mathrm{th}}$. If we define the D2D variability by $R^{\mathrm{D2D}} = \Delta V_{\mathrm{th}}/V_{\mathrm{th}}$, the leakage current is $I_{\mathrm{leak}}^{\mathrm{D2D}} = I_{\mathrm{leak}}\exp\left[-\frac{q\Delta V_{\mathrm{th}}}{kT}\right] = I_{\mathrm{leak}}\exp\left[-\frac{qV_{\mathrm{th}}}{kT}\cdot R^{\mathrm{D2D}}\right]$ and the gate

delay is $T_g^{\mathrm{D2D}} = \left(\frac{V-V_{\mathrm{th}}-\Delta V_{\mathrm{th}}}{V-V_{\mathrm{th}}}\right)^{-\alpha} T_g \approx \left(1+\frac{\alpha}{V-V_{\mathrm{th}}}\Delta V_{\mathrm{th}}\right) T_g = \left(1+\frac{\alpha}{V/V_{\mathrm{th}}-1}R^{\mathrm{D2D}}\right) T_g$, thus the chip frequency is $f^{\mathrm{D2D}} = \left(1-\frac{\alpha}{V/V_{\mathrm{th}}-1}R^{\mathrm{D2D}}\right) f$. When we consider both die-to-die and within-die variations of V_{th}, we obtain a frequency-leakage plot like below.

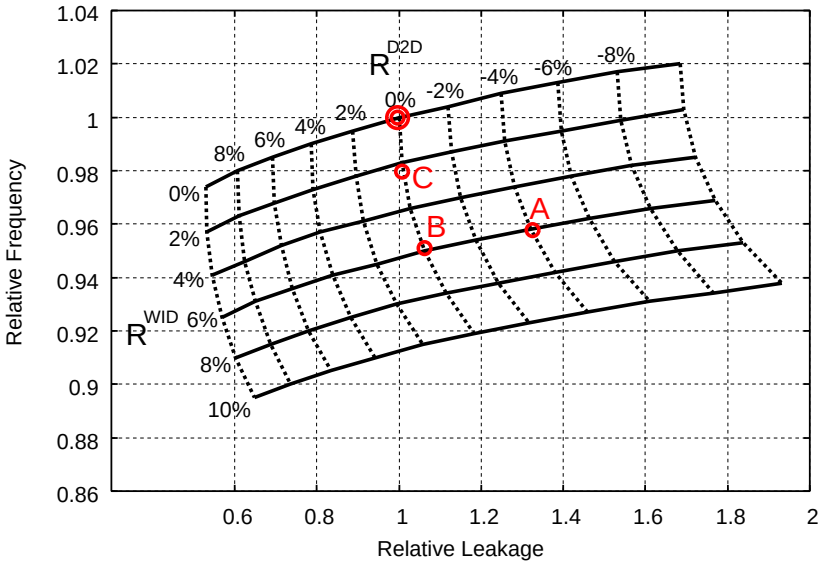

Figure B.2: The effect of die-to-die and within-die variations on the frequency and the leakage

Suppose we are given a die-to-die variation of -4% and a within-die variation of 6% (Point A in Figure B.2). By applying a chip-wide uniform body bias (our fine-grain body biasing can easily do this), we can remove the die-to-die variation (A $\rightarrow$ B). Moreover, we can reduce the within-die variation by applying fine-grain body bias (B $\rightarrow$ C). As it is clear from the figure, coarse-grain body biasing (one body bias per chip) trades off the reduction in leakage against the decrease in frequency. On the other hand, fine-grain body biasing can improve both frequency and leakage, although its effect might be somewhat obscured by the temperature variation.

B.3 Acceptable Region for Chips

We bound the Acceptable Region for the chip in Figure B.3 as follows. The slanted line is given by setting the sum of the dynamic and leakage powers to a constant:

$$K_1 = P_{Dyn} + P_{Leak} = K_2 \times Freq + P_{Leak} \tag{B.3}$$

we can determine K_1 and K_2 if we know, for one of the points in the line, (i) its coordinates and (ii) the ratio between its dynamic and leakage power. We (somewhat arbitrarily) choose the *NoVar* chip to belong to the line — the chip without any process-induced V_{th} variation. For *NoVar*, we know its frequency (6 GHz), and we measure its dynamic and leakage power. Other choices are possible. Finally, we set the horizontal line to be 85% the *NoVar* frequency. We chose this number according to Intel's Xeon processor family at 90nm, which is marketed with a 15% frequency range (3 to 3.6 GHz) [31].

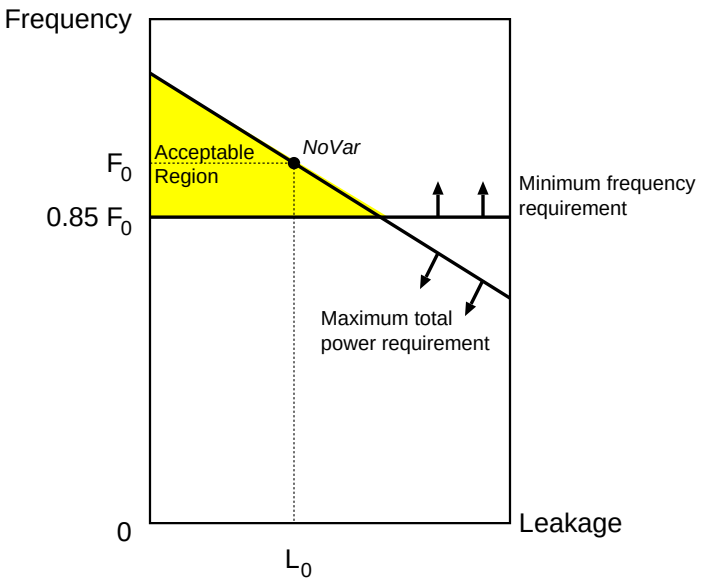

Figure B.3: Acceptable region for chips

References

[1] Apache. http://www.apache.org/.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System*, pages 10–22, 1992.

[3] P. Bernstein. Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer*, 21(2):37–45, 1988.

[4] S. Borkar. Microarchitecture and design challenges for gigascale integration. Keynote in the 37th International Symposium on Microarchitecture, 2004.

[5] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *Proceedings of the 41st Annual Conference on Design Automation*, 2004.

[6] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Conference on Design Automation*, pages 338–342, 2003.

[7] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.

[8] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, 2002.

[9] T. Bressoud and F. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, 1995.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[11] Y. Cao and L. Clark. Mapping statistical process variation toward circuit performance variability: An analytical approach. In *Proceedings of the 42nd Annual Conference on Design Automation*, pages 658–663, 2005.

[12] H. Chang and S. Sapatnekar. Full-chip analysis of leakage power under process variations, including spatial correlations. In *Proceedings of the 42nd Annual Conference on Design Automation*, pages 523–528, 2005.

[13] Z. Chishti and T. Vijaykumar. Wire delay is not a problem for SMT (in the near future). In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 40–51, 2004.

[14] N. Cressie. *Statistics for Spatial Data*. John Wiley & Sons, 1993.

[15] A. Datta, S. Bhunia, S. Mukhopadhyay, N. Banerjee, and K. Roy. Statistical modeling of pipeline delay and design of pipeline under process variation to enhance yield in sub 100nm technologies. In *Proceedings of the Design, Automation and Test in Europe*, pages 926–931, 2005.

[16] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, 2002.

[17] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[18] D. Ernst, K. Flautner, T. Mudge, N. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, and T. Austin. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 7–18, 2003.

[19] D. Frank, R. Dennard, E. Nowak, P. Solomon, Y. Taur, and H. Wong. Device scaling limits of Si MOSFETs and their application dependencies. *Proceedings of the IEEE*, 89(3):259–288, 2001.

[20] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos. Modeling within-die spatial correlation effects for process-design co-optimization. In *Proceedings of the International Symposium on Quality Electronic Design*, 2005.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux kernel behavior under errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 459–468, 2003.

[23] L. Hammond, K. Olukotun, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, and C. Kozyrakis. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, 2004.

[24] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[25] HP Integrity NonStop computing. http://www.hp.com/go/integritynonstop.

[26] HP Integrity servers. http://www.hp.com/products1/servers/integrity/.

[27] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, 2002.

[28] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 169–178, 1996.

[29] IBM System p5 servers. http://www.ibm.com/systems/p/.

[30] IBM Advanced SerialRAID Adapters Technical Reference (SA33-3286-02), September 2000.

[31] Intel Corp. Microprocessor quick reference guide. http://www.intel.com/pressroom/kits/quickreffam.htm.

[32] Iperf Benchmark. http://dast.nlanr.net/Projects/Iperf/.

[33] International Technology Roadmap for Semiconductors (1999) Process Integration, Devices and Structures.

[34] International Technology Roadmap for Semiconductors (2004 Update) Front End Processes.

[35] D. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 86–95, 1993.

[36] A. Journel and C. Huijbregts. *Mining Geostatistics*. Academic Press, 1978.

[37] A. Kahng. The road ahead: Variability. *IEEE Design & Test of Computers*, pages 116–120, May-June 2002.

[38] A. Kahng. How much variability can designers tolerate? *IEEE Design & Test of Computers*, pages 96–97, Nov-Dec 2003.

[39] K. Kanda, K. Nose, H. Kawaguchi, and T. Sakurai. Design impact of positive temperature dependence on drain current in sub-1-V CMOS VLSIs. *IEEE Journal of Solid-State Circuits*, 36(10):1559–1564, 2001.

[40] A. Keshavarzi, S. Ma, S. Narendra, B. Bloechel, K. Mistry, T. Ghani, S. Borkar, and V. De. Effectiveness of reverse body bias for leakage control in scaled dual Vt CMOS ICs. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 207–212, 2001.

[41] B. Kingsbury and J. Kline. Job and process recovery in a UNIX-based operating system. In *Proceedings of the Usenix Winter 1989 Technical Conference*, pages 355–364, 1989.

[42] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, 2003.

[43] T. Kuroda, T. Fujita, S. Mita, T. Nagamatsu, S. Yoshioka, K. Suzuki, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu, and T. Sakurai. A 0.9-V, 150-MHz, 10-mW, 4 mm^2, 2-D discrete cosine transform core processor with variable threshold-voltage VT scheme. *IEEE Journal of Solid-State Circuits*, 31(11):1770–1779, 1996.

[44] C. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, 1992.

[45] D. Marculescu and E. Talpes. Variability and energy awareness: A microarchitecture-level perspective. In *Proceedings of the 42nd Annual Conference on Design Automation*, pages 11–16, 2005.

[46] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the International Conference of Computer Aided Design*, pages 721–725, 2002.

[47] Y. Masubuchi. Personal communication. Oct. 2003.

[48] Y. Masubuchi, S. Hoshina, T. Shimada, H. Hirayama, and N. Kato. Fault recovery mechanism for multiprocessor servers. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 184–193, 1997.

[49] J. Mauro. Tuning Oracle to minimize recovery time. Technical Report 817-4445-10, Sun Microsystems, November 2003.

[50] M. Miyazaki, G. Ono, and K. Ishibashi. A 1.2-GIPS/W microprocessor using speed-adaptive threshold-voltage CMOS with forward bias. *IEEE Journal of Solid-State Circuits*, 37(2):210–216, 2002.

[51] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. In *Proceedings of the European Unix Users Group Spring*, pages 199–206, 1990.

[52] C. Morin, A. Gefflaut, M. Bantre, and A. Kermarrec. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 56–65, 1996.

[53] S. Narendra. Challenges and design choices in nanoscale CMOS. *ACM Journal on Emerging Technologies in Computing Systems*, 1(1):7–49, 2005.

[54] S. Narendra, M. Haycock, V. Govindarajulu, V. Erraguntla, H. Wilson, S. Vangal, A. Pangal, E. Seligman, R. Nair, A. Keshavarzi, B. Bloechel, G. Dermer, R. Mooney, N. Borkar, S. Borkar, and V. De. 1.1V 1GHz communications router with on-chip body bias in 150 nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2002.

[55] S. Nassif. Modeling and forecasting of manufacturing variations. In *Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, pages 145–150, 2001.

[56] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[57] Oracle Real Application Clusters 10g. Oracle Technical White Paper, May 2005.

[58] M. Orshansky, L. Milor, and C. Hu. Characterization of spatial intrafield gate CD variability, its impact on circuit performance, and spatial mask-level correction. *IEEE Transactions on Semiconductor Manufacturing*, 17(1):2–11, 2004.

[59] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, March 2002.

[60] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[61] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002.

[62] The R project for statistical computing. http://www.r-project.org.

[63] A. Reuter. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Transactions on Software Engineering*, SE-6(4):348–356, 1980.

[64] P. Ribeiro and P. Diggle. The geoR package. http://www.est.ufpr.br/geoR.

[65] M. Rosenblum and K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[66] M. Russinovich and Z. Segall. Fault-tolerance for off-the-shelf applications and hardware. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 67–71, 1995.

[67] T. Sakurai and R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, 1990.

[68] Professor N. Shanbhag. Personal communication. ECE Department, University of Illinois, October 2005.

[69] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Western Research Laboratory, August 2001.

[70] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, 2003.

[71] T. Slegel, T. McPherson, J. Navarro, E. Schwarz, K. Shum, C. Webb, R. Averill, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, and J. MacDougall. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[72] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[73] A. Srivastava, S. Shah, K. Agarwal, D. Sylvester, D. Blaauw, and S. Director. Accurate and efficient gate-level parametric yield estimation considering correlated variations in leakage power and performance. In *Proceedings of the 42nd Annual Conference on Design Automation*, pages 535–540, 2005.

[74] B. Stine, D. Boning, and J. Chung. Analysis and decomposition of spatial variation in integrated circuit processes and devices. *IEEE Transactions on Semiconductor Manufacturing*, 10(1):24–41, 1997.

[75] H. Su, F. Liu, A. Devgan, E. Acar, and S. Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 78–83, 2003.

[76] Sun Microsystems. *Sun StorEdge Fast Write Cache 2.0 Configuration Guide*, February 2000.

[77] A. Tanenbaum. *Computer Networks*. Prentice Hall, 2003.

[78] Y. Taur and T. H. Ning. *Fundamentals of Modern VLSI Devices*. Cambridge University Press, 1998.

[79] Transaction Processing Performance Council. Top ten non-clustered TPC-C by performance (as of November 2005). http://www.tpc.org/.

[80] Transmeta. LongRun2 technologies. http://www.transmeta.com/longrun2/.

[81] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid-State Circuits*, 37(11):1396–1402, 2002.

[82] H. Wang, X. Zhu, L. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 294–305, 2002.

[83] WebStone Benchmark. http://www.mindcraft.com/webstone/.

[84] Xeon Nocona prices, November 2005. http://www.newegg.com/Product/.

[85] 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, March 2003.

Author's Biography

Jun Nakano was born in Tokyo in 1965. He received his BS and MS degrees in physics from University of Tokyo in 1988 and 1990, respectively. His master thesis was titled "Quark-Hadron Phase Transitions in the Early Universe." In 1990, he joined Tokyo Research Laboratory of IBM Japan, where he studied queueing theory, combinatorial optimization, and computational geometry. Among them, he particularly likes his work on the minimum cost assignment problem and the Hitchcock transportation problem where efficient application of a randomization technique was made possible via an elegant transformation into equivalent geometric problems. Part of this work was published in SIAM Journal on Computing.

From 1995 to 2000, he had been involved in technical support of IBM's flagship parallel UNIX machines (RS/6000 SP). He worked in various areas ranging from parallelization of scientific and technical applications using HPF, MPI, and OpenMP to commercial benchmarks using parallel databases (DB2 Parallel Edition, Oracle Parallel Server, Informix, etc.). During this period, he coauthored three IBM technical documents (called *Redbooks*) on RS/6000 SP system management, scientific computing on POWER3 processors, and program parallelization using MPI.

In 2000, he was selected for IBM Japan's overseas scholarship program for studying computer science and he has been on leave from IBM since then. He is a recipient of Illiac Fellowship from Department of Computer Science, University of Illinois, from 2000 to 2003.