EXPLOITING APPLICATION LEVEL ERROR RESILIENCE VIA DEFERRED EXECUTION

BY

HENRY JOHN DUWE III

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Assistant Professor Rakesh Kumar

Abstract

Many programs exhibit application level error resilience which allows certain subcomputations to execute in an imprecise, yet energy efficient manner, potentially yielding significant overall energy savings without sacrificing end-to-end quality. In this thesis we identify one fundamental problem that must be addressed to realize these energy benefits: even in applications with a large degree of error resilience, error resilient instructions are interleaved with instructions that must be executed precisely at a fine-grained level (about every seven instructions). This interleaving prohibits any energy savings due to the significant costs associated with switching between the modes, typically via voltage scaling, which may require hundreds to thousands of cycles to transition between levels. We propose a novel execution model for single-core architectures that reduces total switching by deferring execution of instructions requiring mode switches, thereby aggregating instructions of the same energy mode and reducing the number of mode switches. Deferred execution introduces overheads of its own due to data transfer across the two modes, and we present hardware and software optimizations to mitigate these overheads, yielding up to 9%-29% energy savings across a suite of benchmarks.

To my family, for their loving guidance

Acknowledgments

I would like to thank my advisor, Professor Rakesh Kumar, for keeping me focused on the important aspects of this work while driving me to produce quality research. I would also like to thank Sorin Lerner, Ranjit Jhala, Yuvraj Agarwal, and Rajesh Gupta for their helpful comments, insights, and suggestions. Their intellectual rigor and humor were greatly appreciated. Lastly, I would like to recognize all the little bits of vital help provided by the Coordinated Science Laboratory IT staff and other students in my group. I appreciate their attention to my concerns and complaints and their alacrity to brainstorm solutions.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DEQ	Deferred Execution Queue
DEQMU	Deferred Execution Queue Management Unit
DVS	Dynamic Voltage Scaling
RRADW	Robust Read After Delicate Write
DRARW	Delicate Read After Robust Write
DWARW	Delicate Write After Robust Write
DRARR	Delicate Read After Robust Read
DWARR	Delicate Write After Robust Read
RB	Robust Block
EU	Execution Unit
FD	Fetch And Decode Unit
D$	Data Cache
SW	Software
HW	Hardware
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
CAM	Content Addressable Memory

Chapter 1

Introduction

Many programs enjoy a degree of application-level error resilience. For example, in a graphics application, minor errors in the computation of the RGB levels of individual pixels may be imperceptible to the human eye, and hence yield acceptable outputs. Much recent work has investigated the possibility of detecting [1] or specifying such operations [2, 3], and *exploiting* the resilience for obtaining significant energy savings, by executing error resilient operations on low energy (but potentially error inducing) hardware [4, 5].

Not all applications exhibit (enough) exploitable error resilience. Further, as observed previously [4, 3], we find that even in error resilient applications, not all instructions are resilient or *robust* with respect to (errors in) relaxed hardware. Errors in certain critical or *delicate* instructions either cause catastrophic exceptions or led to an unacceptable degradation in output quality. More crucially, we observe that even in resilient applications with a large fraction of robust instructions, there is very often a fine-grained *interleaving* of delicate and robust instructions, yielding a *switch* between the two modes of execution once every seven instructions.

Due to the significant costs of switching between correct (i.e., high reliability) and relaxed (i.e., low reliability) hardware modes, the fine-grained alternation becomes a fundamental obstacle to realizing the energy savings of error resilient applications. For example, using one of the most common mechanisms for trading off correctness and power in relaxed designs, voltage scaling, it may take *hundreds* [6] to *thousands* [7] of cycles to transition between voltage levels depending on whether an on-chip or an off-chip voltage regulator is used. Thus, the cost of mode-switching every seven instructions would dwarf any savings from relaxed execution.

Thus, to realize the potential energy savings from application level error resilience, we need some means of mitigating or eliminating the overhead of fine-grained mode switching. One approach is to radically redesign the

hardware from the circuit level upward, to support fine-grained switching [5].

In this thesis, we present a different, modular and less invasive approach to overcoming the fundamental switching obstacle in the case of single-core execution: a novel execution model that reduces switching by *deferring* the execution of instructions that require a mode switch. The key idea of the execution model is that when instructions requiring a mode switch are encountered, they can be saved or "remembered" for execution later, through the use of a single hardware structure – *a deferred execution queue* – that saves both references to the deferred instructions, as well as their input operands. By deferring the execution of instructions which require mode switches, the queue allows us to *aggregate* sequences of delicate and robust instructions. This reduces the *number* of mode switches and hence, the switching overhead.

If the preceeding description sounds too simple and good to be true, it is. While the queue-based deferred execution model reduces the overhead of switching, the queue itself introduces new overheads which may be prohibitive if implemented naively. Intuitively, to maintain the data dependencies in deferred execution, the queue must also transfer the data operands produced by instructions in one mode and consumed by instructions in the other. In particular, we observe that the nature of communication between instructions of different types, for example, high fan-out and large degrees of data transfer between the instructions of different types, commonly results in prohibitive performance overheads from using a naive queue implementation. Therefore, in order to fully realize the benefits of a deferred execution model, additional software and hardware optimizations will be needed. For this reason, we propose the use of several compiler optimizations and a hardware queue that employs specialized mechanisms for efficiently transferring both data inputs and instruction pointers between modes.

In summary, the thesis makes the following contributions to realize energy savings from application level error resilience [1]:

- We identify two key program characteristics which have a significant impact on the design of systems which exploit application level error tolerance: the interleaving of delicate and robust instructions, and the nature of (data) communicated between the two kinds of instructions (Chapter 2).
- We propose a novel deferred execution model that reduces the overhead of

mode switching in the single core case by effectively aggregating instructions of similar modes. A naive implementation brings its own overheads, which we address through several software and hardware optimizations (Chapter 3).

- We show that with suitable optimizations, deferred execution can yield significant energy benefits (9%-29%) (Chapter 5).

[1] The work presented in this thesis is based on a collaborative effort led by the author. The author wrote the profiling tool that analyzed benchmarks based on their delicate-robust marking. The author also made all energy estimations and gathered needed energy assumptions. The author led and reviewed all manual inspections of benchmarks. The author also led the design of the deferred execution technique and necessary optimizations. Figure 3.1 was not produced by the author. The author participated in the fault injection campaign, but neither wrote the fault injection tool nor performed a majority of the fault injections. Joseph Sloan, Manish Gupta, and Alan Leung aided in manual inspection of benchmarks and design details, produced Figure 3.1, and performed the bulk of fault injections.

Chapter 2

Characterizing Error Resilience

To understand the energy reduction that can be achieved by relaxing hardware correctness, we first need to determine which programs, and which parts of those programs, are even amenable to relaxed correctness. To this end, we performed an empirical analysis of the Parboil [8] and SciMark2 [9] benchmark suites to gather the following information: first, which instructions can or cannot tolerate errors, second, the granularity of interleaving between such instructions, and third, the amount of data communicated between such instructions. We start with some basic definitions for the terminology used in the remainder of this thesis, then describe our characterization methodology (Section 2.1), and finally discuss the empirical characterization of the benchmarks (Section 2.2) in order to motivate the notion of deferred execution.

Definitions. We define the intuitive notions of error resilience as follows. We say an instruction[1] is *robust* if, with high probability, the program completes successfully with acceptable output quality even if the instruction's output contains error. An instruction is delicate if it is not robust. Intuitively, an instruction is delicate if an error in the instruction's output results in abnormal program termination or unacceptable output quality with high probability. We can generalize the aforementioned notions to sequences of instructions. A robust (respectively delicate) *block* is a group of adjacent robust (respectively delicate) instructions within the same basic block. The robust (respectively delicate) *partition* for a program is the set of all robust (respectively delicate) blocks of that program. We further define *unacceptable* program output to be an output with any deviation from the correct result (note that Parboil benchmarks define a range of outputs; SciMark2 benchmarks have unique, discrete outputs). The astute reader will notice that we have yet to define our notion of *high probability*. For ease of presentation, we do so in the current section with a discussion of our experimental

[1] By "instruction" we mean a machine (e.g., x86) instruction

methodology for disambiguating robust and delicate instructions.

2.1 Identifying Robust Instructions

To identify the robust instructions *empirically*, we implemented a tool that uses Pin's binary instrumentation [10] to perform the following experiment for each dynamic execution of the benchmark for a given input set:

1. We randomly select a dynamic instruction, and flip some randomly chosen bit of its output.

2. We record whether that particular execution terminates abnormally or produces unacceptable output.

We repeat the aforementioned experiment, choosing the instruction and flipped-bit from a uniform distribution over the respective universes, to collect statistics of the following form: static instruction I failed $I_N\%$ of experiments. We run enough random experiments for these statistics to stabilize, i.e., for the failure rate of each static instruction to converge to a stable figure. In our experience, the number of experiments required depends on the benchmark. We say that an instruction I *passes with high probability* if it passes at least 80% of the experiments. An instruction which passes with high probability is robust and all other instructions are delicate.

Manual Inspection. To verify that the results of our empirical categorization were accurate, we then manually inspected the assembly code for each benchmark to ensure that the robust instructions were indeed intuitively error resilient. For example, instructions that perform data processing or floating-point arithmetic are typically resilient, but instructions that compute array indices, or pointer targets, or loop bounds are not.

Thus, while we have used a particular, fairly standard, methodology for characterizing instructions as delicate or robust, we believe from the manual analysis that any other methodology would arrive at a partition with similar characteristics.

2.2 Characteristics of Robust Instructions

Next, we describe the results of our classification, in order to obtain a quantitative understanding of the application level error resilience. In particular, our results address three questions. First, how robust are applications – i.e., what fraction of the instructions of a given application are robust? Second, what is the *granularity* of the interleaving between delicate and robust instructions – i.e., how frequently is there a mode switch? Third, how much *communication* is there between delicate and robust instructions – i.e., how much data is transferred across the two kinds of blocks?

Application Robustness. Figure 2.1 shows the fraction of dynamic instructions that were determined to be robust (y-axis) for the suite of benchmarks (x-axis).

Several benchmarks (`sor`, `cutcp`, `stencil`, `fft`, and `LU`) have large fractions of robust instructions (40%-75%), and hence, exhibit a large degree of application level error resilience. Hence, these benchmarks may present an opportunity to save energy by relaxing the correctness of hardware. The common characteristic of these particular benchmarks that leads to a large fraction of robust instructions, is that their kernels are largely dominated by long sequences of floating-point operations. The benchmarks `sgemm`, `sparse`, and `tpacf` have a modest fraction of robust instructions (20%-25%) that may be profitably exploited. These benchmarks tend to have less robust instructions due the presence of more intermediate control operations within inner parts of their kernels. The benchmarks `monte` and `sad` have the smallest number of robust instructions, making them the least likely benchmarks to realize potential energy benefits. Again, a common characteristic is that `monte` and `sad` tend to have more control in the inner parts of their kernel (e.g., updating the state of a random number generator or scanning blocks in an image). See Chapter 5 for more discussion on these particular benchmarks as they are also unique in the type of error-tolerance exhibited.

These results confirm what many other researchers have found, which is that (1) for some applications the fraction of robust instructions may be large enough to actually exploit by running at low-power and conserving energy and (2) not all instructions of applications are robust. For this reason, a classification of the computation as either delicate or robust is an essential part of attempting to reclaim any potential energy benefits.

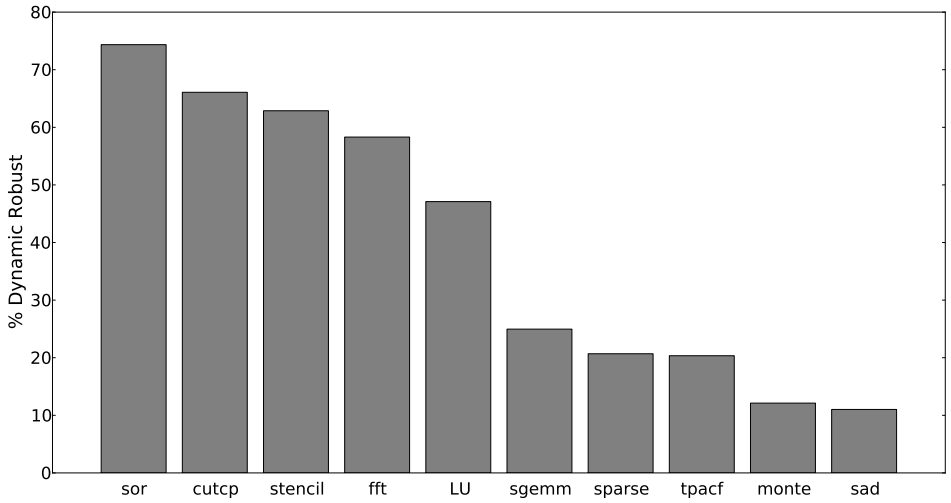

Figure 2.1: **Fraction of Robust Instructions.** Benchmarks are sorted left to right according to the fraction of robust instructions.

Granularity of Delicate-Robust Interleaving. Figure 2.2 shows the average number of instructions *between* high and low reliability mode switches (i.e., the switching period). All benchmarks with the exception of `monte` have an average robust block size of less than 10 instructions. Some of the benchmarks even had average switching periods as low as 2. Therefore, Figure 2.2 suggests that a common program characteristic for many applications is that delicate and robust instructions tend to be interleaved at a very fine granularity. Even in the best case, where the switching period is every ten instructions on average, this still means there will need to be millions to billions of switches in total. This can have a significant impact on the design as discussed in Section 2.3.

Amount of Communication across Delicate-Robust Blocks. Figure 2.3 presents the amount of direct data transferred, via the register file, between the delicate and robust blocks. This *does not* include data transferred through memory accesses. If a value is produced by a delicate instruction and consumed in two different robust blocks, it is counted twice. Figure 2.3 shows that the number of inputs per instruction can vary greatly across the benchmarks. Benchmarks with long chains of robust instructions (e.g., `sor`, `cutcp`, `fft`, and `tpacf`) tend to have a low number of inputs from the delicate partition per instruction on average, while other benchmarks with more control flow and a low number of robust instructions (e.g., `sgemm` and `sad`) tend to have a large number of inputs from the delicate partition

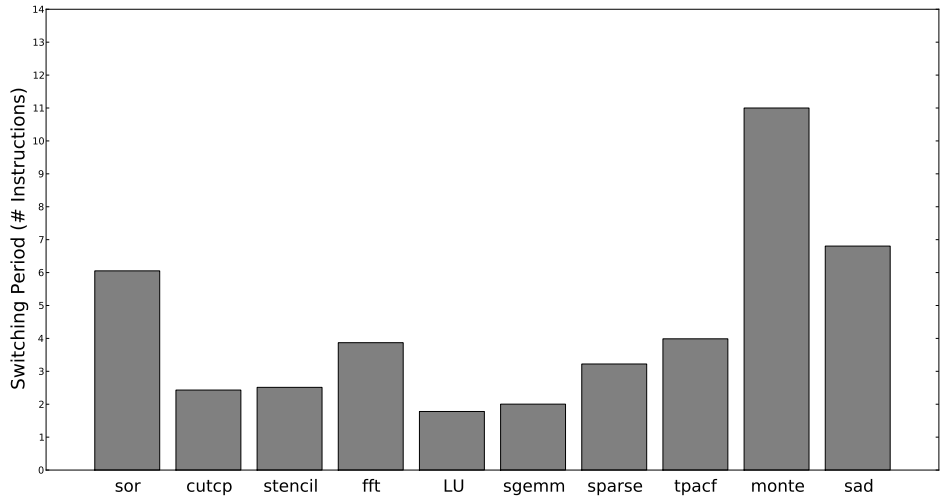

Figure 2.2: **Granularity of Delicate-Robust Interleaving.** The figure shows the average switching period (i.e., the average number of instructions between mode switches).

per instruction on average (>13).

Therefore, the results in Figure 2.3 suggest that a significant amount of data transfer capacity may be needed between delicate and robust partitions. In Chapter 3 and Chapter 5, it becomes quickly apparent that this common type of program characteristic can have a significant impact on our ability to realize any potential energy benefits.

2.3 The Case for Deferred Execution

Given a classification of instructions as delicate or robust, a system can seek to conserve energy by executing delicate instructions in a high reliability mode as usual (i.e., nominal voltage, high power) and robust instructions in a low reliability mode (i.e., low power).

A Naive Approach. One approach for attempting to conserve energy using a single-core architecture would begin by executing delicate instructions in the high reliability mode. Upon *fetching* a robust instruction, the system would trigger a mode switch to the low reliability mode. The core would continue executing robust instructions in this mode until a delicate instruction is fetched, causing a corresponding switch to the high reliability mode.

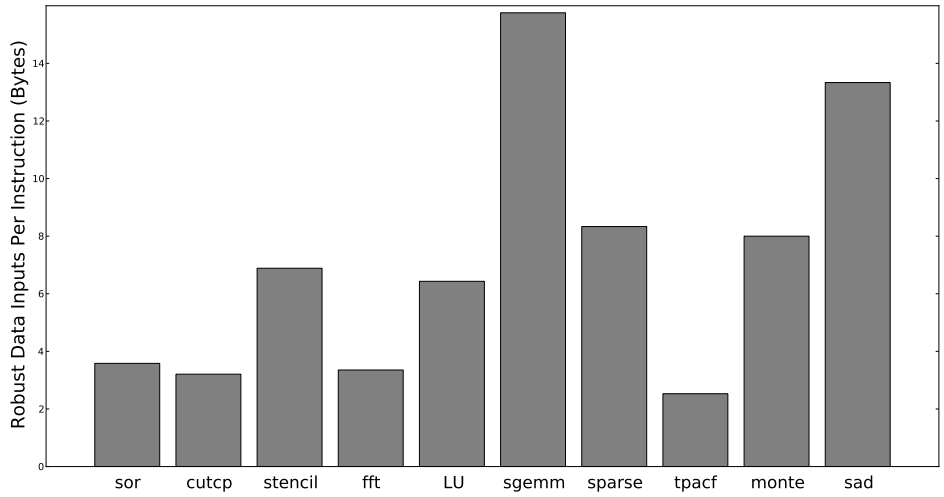

Figure 2.3: **Average Data Transferred across Blocks.** This figure shows the data transferred from delicate to robust blocks *per* robust instruction.

The Cost of Switching. A reliability mode switch entails, at the very least, a voltage scaling of execution units, but may also include a scaling of the entire core (except the instruction cache). The overhead of using voltage scaling at a coarse granularity is a significant limitation encountered by many systems employing Dynamic Voltage Scaling (DVS) – see Chapter 6. In order to protect delicate instructions and their corresponding data, there are also additional overheads from saving and restoring architectural context before switching modes. Thus, assuming any realistic energy overheads associated with switching (100-1000 cycles) and the empirically determined frequency of mode switching in Figure 2.2 (average of seven instructions), the naive approach would result in a catastrophic increase in energy usage over the baseline of executing all instructions in the high reliability mode.

For example, the inner loop of `stencil` is executed 47 million times for the small input, resulting in over 300 million switches in its inner loop. For a switching overhead of 1000 cycles, the switches alone would cause an increase in energy consumption by a factor of over 400x, thereby eliminating any potential savings.

2.4 Software-Based Aggregation

Given that switching costs are of primary concern, we investigate three static techniques for reducing switches between reliability modes: static aggregation, loop unrolling, and mode promotion. In short, all three techniques seek to produce long spans of robust instructions by exploiting three characteristics of robust code.

Static Aggregation. Static aggregation is a form of code motion that relocates robust instructions within a basic block into a continuous span, with the constraint that no data dependencies from the original program can be violated. This optimization relies on the fact that robust instructions often only depend on the results of other robust instructions and thus can be hoisted away from the delicate portion of a basic block.

Loop Unrolling. In cases when a loop body is robust, loop unrolling followed by static aggregation allows us to produce even larger robust blocks by aggregating robust instructions across several iterations of the loop. This optimization capitalizes on the fact that robust instructions tend to be the data processing instructions in the body of the innermost loops of computational kernels.

Mode Promotion. Finally, mode promotion opportunitistically assigns some robust instructions to be executed in *high reliability mode*, despite their classification as robust, if the mode switch to *low reliability mode* is likely to waste more energy than it saves. In particular, we apply the heuristic that a mode switch is ignored when a robust instruction's output flows to a delicate instruction within the same basic block, because we would immediately switch to high reliability mode to execute the delicate instruction anyway.

From our experiments, discussed more in Chapter 5, we find that static aggregation and loop unrolling provide on average only a 2x-3x reduction in the total number of mode switches during the execution of the program. While beneficial, this is not nearly enough reduction to have an effect on the 1000%-10000% switching overheads from the naive approach. Although we investigated mode promotion on all benchmarks, we found that only `cutcp` and `sad` showed any benefit from this technique and thus did not investigate it further.

Chapter 3

A Deferred Execution Model

Given the astronomical costs of the naive approach, even with static aggregation, it is essential to develop mechanisms to reduce the switching overheads. At a high level, instead of immediately switching modes on fetching the first robust or delicate instruction, we instead *defer* robust instructions to a queue to keep track of the robust instructions that have yet to execute. We execute robust instructions only when we *must*: either to respect a flow dependency in the original program, or because the queue is full and must be emptied to make forward progress. In this way we can batch the execution of robust instructions, reducing switching costs.

Consider the example shown in Figure 3.1, which contains a sample program consisting of two delicate (I1, I3) and two robust instructions (I2, I4), as shown in green and red, respectively. To provide some intuition, we will now walk through each step of deferred execution for this example.

1. Delicate instruction I1 executes in high reliability mode. The output i is computed and updated precisely ($i = 201$). The next instruction, I2, is robust, so it is deferred to the queue, as depicted by the red arrow labeled 1. At this point, the queue holds a pointer to instruction I2, as well as the data value $i = 201$. Note that no mode switch has taken place yet.

2. Instruction I3 is delicate and executes in high reliability mode, which updates the value of i to 202. Instruction I4, however, is robust and is deferred to the queue, as indicated by the arrow labeled 2. The queue is updated with instruction pointer I4 and data input $i = 202$.

3. At this point, we have executed all the delicate instructions and must switch to a low reliability mode to begin executing the deferred robust instructions. The instruction pointer for I2 and data input $i = 201$

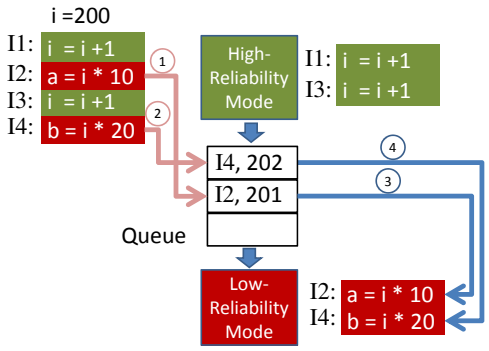

Figure 3.1: **A Deferred Execution Model Example.**

are read from the queue, and I2 is executed in low reliability mode, as shown by the blue arrow labeled 3.

4. Finally, the pointer for I4 is read off the queue, along with input data $i = 202$, and I4 executes in low reliability mode as depicted in the figure as blue arrow 4.

3.1 The Deferred Execution Queue

There are two requirements that our queue must fulfill. First, any dependencies between the robust instructions it contains must be preserved. Second, it must provide reliable storage and access for any values produced by delicate instructions since these values, by definition, cannot tolerate errors. We first present an abstract queue design based on these requirements, and then propose the additional hardware and software support for it. We illustrate this support with a `sgemm`-like kernel shown in Figure 3.2.

To defer a robust block for future execution, we record a partial snapshot of its context in the queue in which it would have executed in the original program. This context consists of the starting address of the robust block, and the set of input values consumed by that block. Specifically, we enqueue the starting address (PC) of the robust block, followed by a sequence of input values, corresponding to the register contents produced by delicate instructions that the robust block consumes. In this way, one can view the queue as a sequence of *activation records*, where the first element of each

```
        movaps 8(%rsp), %xmm3
.L1:    add %r12, %rax
        xor %r9, %r9
.L2:    add #1, %r9
.P1:    movaps %xmm3, %xmm0
        addss (%rax, %r9, 8), %xmm1
        mulss %xmm0, %xmm1
        cmp %r9, %r12
        jle .L2
        add #1, %r10
.P2:    addss %xmm1, (%r13, %r10, 8)
        cmp %r10, %r11
        jle .L1
        push %r13
        call <print_output>
```

Figure 3.2: **Original x86 for a `Sgemm`-Like Kernel.** Gray background indicates robust instructions; blocks labeled `.P1`, `.P2` are robust.

record is always a starting address, and the remaining elements are input values defined by delicate instructions.

Robust Input Data. Through significant inspection (both manual and automatic) we see that the vast majority of data inputs to the robust partition are addresses or parts of addresses (indices, etc.). Consequently all data inputs passed on the queue are addresses. In the rare instance that a value stored in a register must be passed from the delicate partition to the robust partition, a new robust block is generated which simply moves the value from the queue to a register. Instructions are inserted into the delicate partition to enqueue the values onto the queue and then to enqueue the new robust block address. R3 in Figure 3.3 is an example of such a robust block. Once that data is in a register in the robust partition it never needs to be re-transferred again.

Flushes. To execute robust instructions, we first switch the processor state into a low reliability mode, then restore and execute each robust block from the queue in turn until it is empty (i.e., *flush* the queue). A queue flush is trigged under three conditions. First, when the queue fills up it needs to be flushed to make forward progress. Second, on program termination any remaining queue entries are flushed. Third, if a delicate instruction has a memory data dependency on a robust instruction in the queue, then the queue must be flushed before the delicate instruction may execute. Note that this is done for memory dependencies, not register dependencies, as all register dependencies are implicitly preserved by virtue of the order of the queue. We incorporate an additional instruction, `flush`, to allow for a

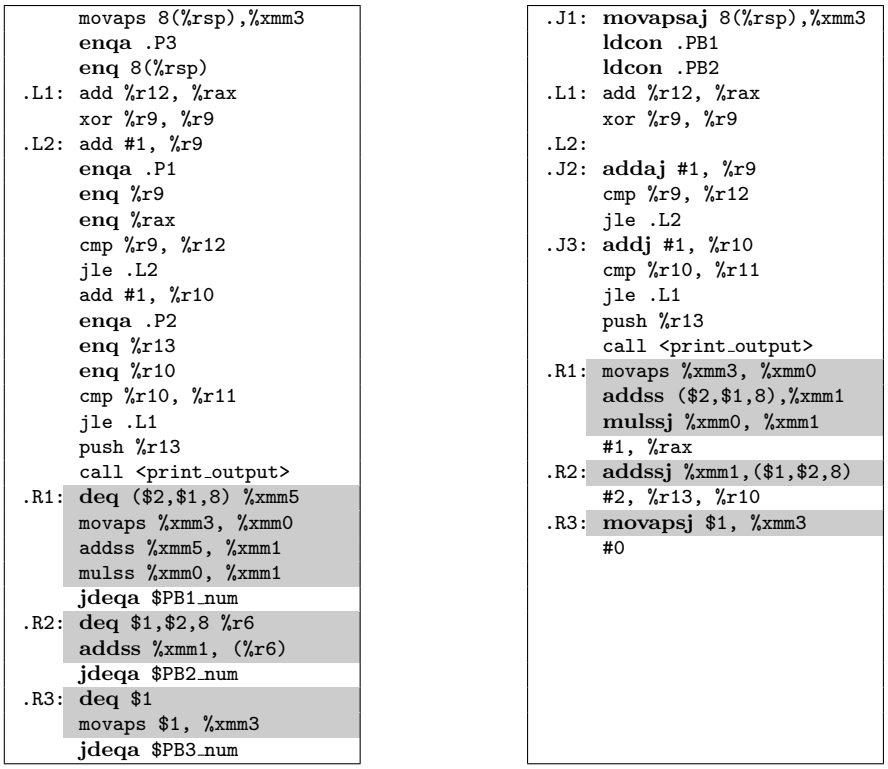

Figure 3.3: **Deferred Execution Assembly Modifications.** (a) `Sgemm`-like assembly code snippet transformed to utilize deferred execution using only basic hardware-support. The robust blocks have been relocated and an `enqa` instruction is used to enqueue its address during delicate execution. Inputs to the robust blocks is communicated via the `enq` and `deq` instructions. (b) `Sgemm`-like assembly code snippet transformed to utilize deferred execution using additional hardware support for data and control flow passing. Modifications to the ISA appear in **bold**.

manual queue flush for the third case.

Memory Data Dependencies. There are four classes of dependencies through memory, which correspond to the various permutations of reads and writes between delicate and robust blocks: dependencies within robust blocks (robust-robust), within delicate blocks (delicate-delicate), from robust to delicate blocks (robust-delicate), and from delicate to robust blocks (delicate-robust). Of these four classes, only robust-delicate dependencies require the use of a flush, while the rest require no special handling.

A common delicate-robust dependency is a robust read after a delicate write (RRADW), which occurs, for example, when delicate code initializes

an in-memory array from disk for further processing by robust code. Delicate-robust dependencies are respected automatically in the proposed execution model because robust blocks only execute after being enqueued. Similarly, delicate-delicate dependencies are respected because we never modify the ordering of delicate instructions with respect to each other. Robust-robust dependencies are also respected because robust blocks are executed in queue order, which preserves the original program ordering. The last class of dependencies, those from robust to delicate instructions, are potentially problematic, so we handle them carefully.

Robust-Delicate Dependencies. DRARW dependencies (true dependencies) are handled by requiring a manual flush immediately before the delicate read instruction executes. DWARW dependencies (anti-dependencies) also require explicit flushes to prevent the robust write instruction from executing after the delicate write instruction. Such dependencies happen very rarely in practice. More subtly, even read-only dependencies (DRARR dependencies) must be handled specially; low reliability mode operations may in fact lower cache voltages to a point where a robust read could corrupt the stored data value. Thus a delicate instruction reading that same value after the execution of the robust read would potentially read an incorrect value in high reliability mode! This is also a very rare occurrence and could be protected either by using a small amount of duplication on such data. Finally, DWARR dependencies are allowable, as the delicate write clobbers any potential errors introduced by the robust read, but a manual flush is required to ensure the robust instruction reads the correct value.

3.2 Basic HW Support for Deferred Execution

A possible implementation for our deferred execution would use a software-implemented queue with only minimal hardware modifications: an ISA extension for an instruction that initiates the mode switches. Such a system would need software support for enqueuing and dequeuing data, with each such operation taking at least four additional instructions: one fetch and update for the queue head (tail), two to check bounds and branch beyond the mode switch instruction, and one to load/store the input itself. These overheads are so unreasonably high that it is clear we simply cannot implement

our deferred execution model without additional architectural support.

Hardware-supported dynamic aggregation incorporates a reliable hardware implementation of the *deferred execution queue* to hold addresses for robust blocks and memory accesses. We describe the basic hardware support required for deferred execution next.

DEQMU. We introduce the deferred execution queue management unit (DEQMU), a reliable hardware unit that manages the queue. The DEQMU holds the queue state (e.g., head, tail, etc.) and implements the logic to dequeue and use addresses (effectively memory indirect operations). The DEQMU also performs the robust block address dequeue to PC instruction. The deferred execution queue (DEQ) is a hardware data store accessed exclusively through the DEQMU. While the DEQ incurs a power overhead (details in Section 4.2), it facilitates the deferred execution model resulting in net energy savings.

The DEQ is implemented as a two-level structure. A small (less than 1KB), low-latency buffer holds the current robust block's queue elements and is backed by a larger, low-power, higher delay store that holds all remaining queue elements. When one robust block's queue elements are dequeued the next robust block's queue elements are loaded out of the second level store into the buffer. Besides the head of the queue (the current robust block's address) the remainder of the buffer stores the inputs to the robust block. When a dequeue instruction executes, a reliable address is used to perform a memory access. Note that only the storage and passing of the address to the memory needs to be reliable. Any storage or communication of values can be unreliable.

New Instructions. The proposed hardware support requires several modifications to the ISA. An `enqa` instruction is used in the low reliability mode to enqueue a robust block address onto the queue. A `jdeqa` instruction reliably increments the head of the queue by an immediate value and loads the address at the new head of the queue into the PC. To communicate inputs to the robust blocks which are generated by delicate blocks, the `enq` and `deq` instructions are used to enqueue and dequeue the inputs respectively.

Figure 3.2 illustrates these modifications using an example code snippet (from an `sgemm`-like benchmark). The example code has four robust instructions and two robust blocks (after applying static aggregation). To execute

this code in a deferred manner, using the DEQ, each robust block is relocated to the end of the listing and replaced with an `enqa` instruction. Additionally, after enqueing the robust block address, we then use several `enq` instructions to communicate the inputs to the robust blocks. These transformations are shown in Figure 3.3a.

Reliable Effective Address Calculation. Since almost every modern processor, whether load-store, RISC or CISC, requires an effective address calculation for each memory address, our design includes reliable hardware adders and a shifter in the load-store unit to compute the addresses reliably. Immediate scales and displacements come from the fetch portion of the instruction and are either reliable (mild and medium design points–see Section 4.2) or we accept the low probability that they will be corrupted (aggressive design point).

3.3 HW Support for Direct Queue Access for Data Passing

Despite providing the basic hardware support for the deferred execution queue described in Section 3.2, we observe that there is significant performance overhead for communicating inputs to robust blocks via the queue (i.e., multiple `enq` and `deq` instructions). This is also clearly evident from the doubling of instructions in the inner loops of the code in Figure 3.3a. For this reason, we propose the use of additional hardware support that allows delicate instructions that produce inputs for robust instructions, to enqueue those values directly onto the deferred execution queue without any additional performance overhead. Thus, any instruction that may be marked as delicate and generate a result used by a robust instruction must have an analogue that both writes its result to the destination register and enqueues its result on the DEQ (e.g., an `adda` instruction).

Robust Block Contexts. While profiling benchmarks we observe that certain dynamic delicate instructions produce an output value used by multiple dynamic robust blocks; we refer to this instruction as having a high fan-out. The deferred execution queue requires that each robust block has this value enqueued for it. Typically this would require additional instructions executed

in the high reliability mode to enqueue these values from registers for each robust block. This solution incurs significant energy overheads from fetching and decoding additional instructions. To reduce this overhead, we propose a robust block context unit whose purpose is to hold the register context for a particular robust block. Registers that are required for a robust block are statically known and loaded into the context unit before the robust block becomes active. In the high reliability mode, this unit snoops on the common data bus and updates the value of any register in an active robust block's context. Each context is a small CAM addressed by register ID. This unit does not need to be very large because few robust blocks are generally active at the same time. Furthermore each robust block generally only requires a few registers to be tracked in this manner.

The `ldcon` instruction loads the robust block's context into an available context structure. `Enqa` instructions enqueue the values stored in the robust block's context and also check whether or not a reliability mode switch is triggered.

3.4 HW Support for Direct Queue Access for Control Passing

Communicating the addresses of robust blocks onto the queue (i.e., the control flow of the robust blocks) can also be a significant performance overhead which prohibits us from realizing energy benefits. Recall that whenever a robust block is encountered, the address for that block is enqueued so that it can be executed later after more robust instructions have been aggregated. We propose hardware support that allows the robust block pointers to be written and read from the queue in an intuitive fashion that eliminates the need for any special enqueue and dequeue instructions and their associated performance overhead.

Similar to hardware support for direct data passing to the queue, hardware support for control flow passing also entails incorporating new instructions which are the analogue of any instruction which immediately precedes a robust block (e.g., an `addj` instruction). An `addj` instruction in the high reliability mode computes an add and then enqueues the address of the deferred robust block onto the queue without incurring any additional performance

overheads since the enqueue occurs in parallel and its completion rarely required for continued delicate execution. Likewise one of these "j" instructions is the last instruction of a robust block. An `addj` instruction in the low reliability mode computes an add, dequeues the address of the next robust block, and loads the dequeued value into the PC. Note that the dequeue happens in parallel and is read from the low-latency DEQ buffer.

The support for these additional instruction basically involves including a new bit (*switch bit*) for each instruction. The *switch bit* is set to zero, except on the last instruction which is executed before a high to low reliability mode switch or on the last instruction of a robust block. In other words, the *switch bit* signals to the system that a switch is needed from one of the two modes. If the switch bit = 1, and the system is in the high reliability mode, the delicate instruction enqueues a robust block address RX onto the queue. The RX value is loaded from a table which maps PC values to robust block addresses (for all instructions with switch bit = 1). If the switch bit = 1, and the system is in the low reliability mode, the robust instruction gets the next PC value from the queue.

Figure 3.3b continues with the example from Figures 3.2 and 3.3a, and shows how the program would be transformed to utilize the additional hardware support for direct data and control passing in the queue. The instructions which contain a mode switch after the execution (i.e., *switch bit* = 1), use an instruction ending with "j" (e.g., "addj"). Delicate instructions which generate inputs for robust instructions are signified with "a" (e.g., adda). Using additional hardware support for directly passing data and control flow to the queue, clearly reduces the number of cycles needed for communicating data between the delicate and robust partitions significantly. In fact, in Chapter 5, we observe that these modifications are essential to realizing any potential energy benefits.

Table 3.1 summarizes the additional instructions required for each level of hardware support. Effectively three bits are required for the proposed hardware support. One for direct data enqueing, one for direct data dequeueing (i.e., addition of the queue addressing mode), and one for direct robust block enqueueing and dequeueing.

Table 3.1: **Added Instructions.** Each type of added instruction is listed along with its function and which level of hardware support requires it.

Added Instructions	**Function**	**Hardware Support Level**
`enq`, `deq`	Enqueues and dequeues data to and from the DEQ	Basic
`enqa`, `jdeqa`	Enqueues robust block addresses on DEQ and dequeues robust block addresses into PC	Basic, Direct Data
Data Producing (e.g., `adda`)	Performs normal operation, also enqueues result on queue	Direct Data, Direct RB Address
Data Consuming (e.g., `addss ($2,$1,8), %xmm1`)	Dequeues input operands from queue (using immeditate indices), performs normal operation	Direct Data, Direct RB Address
Control Passing (e.g., `addj`)	Perform normal operation then enqueue robust block address	Direct RB Address
`ldcon`	Loads the static robust block contexts for specified robust block	Direct Data, Direct RB Address

Chapter 4

Methodology

We start by describing the different architecture– and execution–specific parameters that we use to evaluate the energy savings (Section 4.1). Next, we describe various architectural assumptions and how we use them to arrive at values for the architecture-specific energy parameters (Section 4.2). Finally, we describe the experimental setup used to determine the values of the execution-specific parameters, and hence the overall energy savings (Section 4.3).

4.1 Energy Models

We calculate the total energy required (to execute a particular benchmark) from a set of architecture- and execution- specific parameters. We obtain values for the former by an analysis of the energy required by different hardware components, and the latter from statistics gathered from the execution of the benchmark. Table 4.1 and Table 4.2 enumerate these parameters.

Table 4.1: **Architecture-Specific Parameters.** We abstract energy costs of the underlying architecture into these parameters.

E_{hi}	The energy required to execute a single instruction in the default, high reliability, high energy mode
E_{lo}	The energy required to execute a single instruction in the low reliability, low energy mode
E_{sw}	The energy required to carry out a single mode switch across high and low reliability modes
E_{Q}	The fixed energy required by the deferred execution queue for the duration of a single instruction
E_{qop}	The additional energy required to execute a single queue operation (i.e., an `enq` or `deq`)

Table 4.2: **Execution-Specific Parameters.** We summarize the characteristics of each program execution with these parameters.

N_{Del}	The total number of (dynamic) delicate instructions in an execution
N_{Rob}	The total number of (dynamic) robust instructions in an execution
N_{Sw}	The total number of mode switches in an execution
N_{CySw}	The number of cycles for a mode switch
N_{Qop}	The total number of deferred execution queue (e.g., `enq` or `deq`) operations in an execution

Energy Models. We combine the architecture– and execution–specific parameters to obtain three models for energy costs, depending on how the program is executed. First, the baseline, where all instructions execute at the high energy level

$$\mathbb{E}_{Bas} \doteq (N_{Del} + N_{Rob}) \times E_{hi}$$

Next, an *ideal* model where the robust instructions execute with low energy without any further overheads

$$\mathbb{E}_{Opt} \doteq \mathbb{E}_{Bas} + N_{Rob} \times (E_{lo} - E_{hi})$$

Next, a naive model where robust instructions execute in the low energy level, and a mode switch is triggered whenever one is encountered and whose costs are accounted for

$$\mathbb{E}_{Nai} \doteq \mathbb{E}_{Opt} + N_{Sw} \times E_{sw}$$

Next, the deferred execution model where we account for both the fixed cost of the queue and each queue operation

$$\mathbb{E}_{Def} \doteq \mathbb{E}_{Nai} + (N_{Del} + N_{Rob} + N_{Sw} \times N_{CySw}) \times E_Q + N_{Qop} \times E_{qop}$$

Finally, the deferred execution model with hardware support for direct data robust block transfers has, in essence, N_{Qop} is 0 as the robust inputs are enqueued directly by the delicate instruction which defines the input and the robust block addresses are enqueued by the instruction that is executed immediately before them (marked with an "a" and "j" suffix respectively–see Section 3.3)

$$\mathbb{E}_{Def+} \doteq \mathbb{E}_{Def} - N_{Qop} \times E_{qop}$$

Effect of Optimizations on Execution Parameters. Aggregating instructions via static analyses (e.g., by loop unrolling) may marginally lower N_{Sw}. Aggregation with static approaches are limited beyond the basic block level. Using deferred execution reduces the number of switches N_{Sw} roughly by a factor of one over the number of robust blocks that can fit on the queue *at a time*. The number of robust blocks that can fit on a queue at one time depends on how many times each robust block is executed and the number of data inputs that the block requires to be on the queue. Furthermore, we also account for robust flushes that can drain the queue and hence, limit the number of robust blocks that can be enqueued (and thus increases N_{Sw}).

4.2 Architectural Energy Models

As described in Section 4.1, our energy model accounts for energy at the instruction level. We reflect lower-level energy costs at the instruction level by using an energy estimation similar to that in [3].

Concretely, we assume that at the high reliability level, each instruction takes one unit of energy and one unit of time to execute (i.e., $E_{hi} = 1$). Thus, each additional instruction added due to deferring incurs an overhead of one energy unit and one time unit. To evaluate energy savings, we assume that each instruction executed in the low energy level takes a fraction of a unit (i.e., $E_{lo} < 1$). We consider such assumptions reasonable as even between floating-point and integer instructions, there is only approximately 8% energy difference for their entire execution [3].

Next, we describe the relative energy costs of key architectural elements, and discuss how we aggregate them into three values for E_{lo} corresponding to

mild, medium or aggressive settings for the low energy, low reliability mode.

Energy Costs: Core. We use the core model of [3] which assumes the core accounts for 65% of energy of a processor with the cache using 35%. Execution functional units account for 40% of the energy of the core. We assume that the core may be voltage scaled to save up to 40% energy. We performed a detailed measure of process, voltage, and temperature (PVT) variation guardbands across a set of open cores and determined that the mean worst-case guardband corresponds to 40% energy savings. The cache energy may be reduced by 70% (the mildest assumption used by [3]).

Energy Costs: Data Cache. A potential problem may arise when caches containing delicate data are voltage scaled while executing robust instructions. The only data that will have an increased susceptibility to corruption are those that reside in cache lines accessed (read or written) at reduced voltage. Thus if the robust and delicate data can be partitioned and there are no DRARR data dependencies (DWAR* dependencies overwrite the data anyway and DRARW dependencies already must accept the possibility of data corruption), the data cache may be voltage scaled. By manual examination we have found that these two conditions are met for our benchmarks.

Energy Costs: Queue. To estimate the energy overhead from the queue structure, the low-latency buffer portion of the queue is assumed to be implemented using flip-flops (a la registers). The large data store of the queue is assumed to have similar energy to a cache, but implemented with lower power SRAM cells [11, 12, 13]. We estimate that the queue store requires an energy overhead of 10% of a similarly sized cache. Lastly we assume that the queue access latency is the same as a register access since most accesses come directly from the low-latency buffer structure.

Energy Levels: Mild, Medium and Aggressive. We evaluate deferred execution under three different energy-reduction design assumptions–mild, medium, and aggressive–which correspond to three different valuations for E_{lo}. *Mild* energy reduction only uses voltage scaling on functional units in the execution stage of the pipeline. In particular, mild assumes a 40% reduction of power for only execution units. *Medium* energy reduction uses voltage

Table 4.3: **Three Relaxed Hardware Correctness Design Assumptions for a Single Core Architecture.** EU is Execution Units, D$ the L1 Data Cache, FD is Fetch and Decode Units.

Design	**Relaxed Components**	E_{lo}
Mild	E	0.896
Medium	EU, D$	0.651
Aggressive	EU, D$, FD	0.495

scaling on the execution functional units and the L1 data cache. Specifically, medium assumes the mild reduction and a 70% reduction in energy used by the data cache. *Aggressive* energy reduction uses voltage scaling on the *entire* core and the L1 data cache. The three design assumptions are visualized and tabulated in Table 4.3.

4.3 Experimental Setup

The last piece of the methodology is the set of benchmarks that we used to evaluate savings, as well as the means by which we compute the values of the different execution-specific parameters, such as the number of delicate and robust instructions (N_{Del}, N_{Rob}), the number of mode switches (N_{Sw}), and queue operations (N_{Qop}.)

Benchmarks. We evaluate deferred execution on a set of applications drawn from the SciMark2 and Parboil benchmark suites, which include scientific and multimedia applications. We selected these benchmarks as prior work [14, 3, 5] has demonstrated that they exhibit a significant amount of application level resilience, and hence, are likely to be suitable candidates for energy reductions. Each application was compiled for an x86-64 architecture with `gcc` using optimization level `-O2` and debugging information.

Simulating Executions. Potentially robust instructions are marked at the x86 assembly level using the categorization described in Chapter 2. As other research has shown [14] and our fault injections confirmed, almost all address calculations must be delicate. We make the conservative assumption that control flow in the program is also likely delicate and execute it in the high reliability mode, based on prior research [14]. Another Pintool profiles

each benchmark to gather data on the dynamic interleaving of robust and delicate instructions. It also returns various counts on instructions, robust instructions, number of executions of individual robust blocks, and amount of data directly crossing the delicate- robust boundary. From these, and the sizes of the deferred execution queue, we can derive the values of N_{Del}, N_{Rob}, N_{Sw} and N_{Qop}. When we evaluate the benchmarks, e.g., to determine the delicate and robust instructions (Chapter 2) or to compute the energy savings (Chapter 5), we *only* consider the "kernel" portion of these applications. That is, the Parboil and SciMark2 timing code is ignored to ensure that it does not skew the results.

Manual Inspection. We perform a manual inspection of assembly code to determine if any basic static aggregation or loop unrolling applies to each robust block. If such static aggregation can be applied, the number of robust addresses enqueued for that robust block is reduced, reducing both the performance overhead from executing `enqa` and `jdeqa` instructions (i.e., N_{Qop}) and the reduction of mode switches (N_{Sw}). The manual inspection also yields the locations of robust flushes.

Chapter 5

Results

Next, we present the overall energy savings from the deferred execution of robust instructions, and give a detailed breakdown of the benefits yielded by each of the optimizations described in Chapter 3, in the context of a single core architecture. In particular, we demonstrate how our optimizations affect the number of switches, allowing energy benefits. Finally, we explore the sensitivity of deferred execution energy reduction to the switching overhead and queue size.

Overall Energy Savings. Table 5.1 summarizes the overall energy reduction of the naive architecture, where instructions are executed as compiled and the system switches modes whenever the instruction types change, and the deferred execution model, using different types of hardware and software support. Savings are shown as percentages against the baseline where all instructions execute at E_{hi}. The first observation (row 1) from the results of Figure 5.1 is that with the naive approach the overhead of switching more than erases any hope of recovering the potential energy benefits, and yields a dramatic *increase* in energy consumption (as indicated by the large negative savings percentage).

Static aggregation (row 2), e.g., loop unrolling, lowers the the number of switches N_{Sw} enough to reduce the energy overhead on average by a factor of two. Therefore using static approaches alone is clearly not enough to reduce the large increase in energy consumption from frequently switching.

Next, observe that while the deferred execution model reduces the overhead of switching (by lowering N_{Sw}), it also introduces *new* overheads from transferring state and data to the queue. Deferred execution using software support alone (row 3) lowers N_{Sw} enough to provide another factor of 100 improvement in energy consumption, but is still insufficient to achieve overall energy reductions. By using the basic hardware queue support for deferred execution, (row 4) the cost of queue operations is reduced (lowering E_{qop}) and

Table 5.1: **Average Energy Savings for Various Design Points.** The savings are with respect to the baseline $\mathbb{E}_{Bas}$ with all instructions run in high reliability mode. Only a design with both deferred execution and hardware support for writing all elements directly to the queue achieves average energy benefits. A negative number indicates an *increase* in energy over the baseline.

Technique	**Cost**	**Savings (%)**
Naive	$\mathbb{E}_{Nai}$	-32718
Static	$\mathbb{E}_{Nai}$	-12331
Dynamic (SW)	$\mathbb{E}_{Def}$	-158
Dynamic (HW)	$\mathbb{E}_{Def}$	-32
Dynamic ("" + direct data)	$\mathbb{E}_{Def}$	-3
Dynamic ("" + direct RB address)	$\mathbb{E}_{Def+}$	10

hence, brings the energy increase down to 32%. Next, by adding hardware support for direct data addressing (row 5), we reduce the number of queue operations N_{Qop}, and this allows for the deferred execution to nearly break even on energy with the baseline. Finally, with hardware support to directly populate all queue elements, both data and robust block addresses (row 6), we can eliminate N_{Qop} entirely. This enables deferred execution to achieve positive energy reduction relative to the baseline single-mode execution.

Energy Savings Per Benchmark. Figure 5.1 shows the energy reduction for each benchmark given a switching overhead of $E_{sw} = 1000$ and a queue size of 64KB. The switching overhead is chosen to reflect a realistic energy overhead for switching a large fraction of a chip. A queue size of 64KB represents the best queue size for the largest number of benchmarks given the switching overhead of 1000. We discuss the impact of queue sizes on the the number of switches N_{Sw} later. The three graphs correspond the aggressive (left), medium (center) and mild (right) energy settings. In each case, the y-axis, shows the ideal energy savings ($\mathbb{E}_{Opt}$) assuming no overheads at all, and the energy saved by deferred execution assuming all optimizations ($\mathbb{E}_{Def+}$). Both set of results are shown as percentages relative to the baseline ($\mathbb{E}_{Bas}$). The benchmarks are sorted by decreasing fraction of robust instructions. Note that a positive energy reduction is desirable while a negative energy reduction indicates a case where overheads from optimizations actually increase the energy of a benchmark.

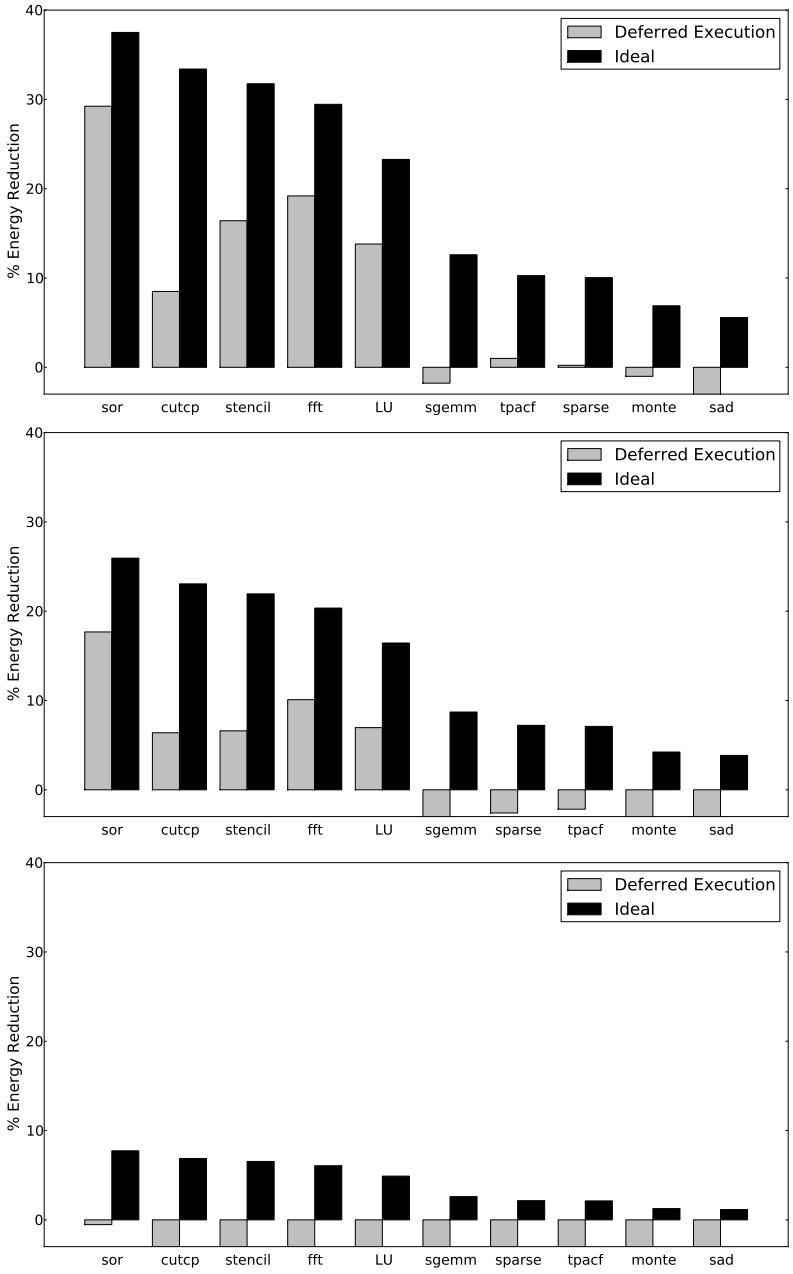

Figure 5.1: **Energy Savings per Benchmark with (T) Aggressive, (M) Medium and (B) Mild Assumptions.** For each benchmark, we show energy savings from deferred execution ($\mathbb{E}_{Def+}$) and ideal savings ($\mathbb{E}_{Opt}$). Benchmarks fall into three classes: (1) those with a large fraction of robust instructions, (2) those with a low fraction of robust instructions, and (3) those with delicate-robust dependencies in their innermost loop. In the *aggressive* setting (top), benchmarks in class 1 get significant energy savings (9%-29%), those in class 2 get limited savings (<5%), and those in class 3 may see large energy increases ($\gg$100%). With the *medium* assumptions (middle) only the benchmarks in class 1 see savings, albeit reduced to 5%-20%. There are no savings in the *mild* setting (bottom).

Savings with Aggressive Energy Settings. For an aggressive design where all logic units and the data cache are mode switched (Figure 5.1(left)) we find three classes of benchmarks:

- Class 1: benchmarks `sor`, `cutcp` `stencil`, `fft`, and `LU` where deferred execution achieves significant energy benefits (10%-30%) due to a large fraction of robust instructions and rare delicate-robust dependencies.
- Class 2: benchmarks `sgemm`, `sparse`, `tpacf`, and `monte` where deferred execution achieves limited (<5%) or no energy benefits due to a low fraction of robust instructions.
- Class 3: benchmark `sad` where deferred execution offers no energy benefits due to frequent delicate-robust dependencies regardless of fraction of robust instructions.

Monte Carlo simulations have been identified as having a significant fraction of robust operations [3], yet our results indicate that `monte` shows a limited fraction of robust instructions (12%). The reason for the lack of robust instructions is that the pseudorandom number generator used requires both significant amounts of control and many address computations to update its state. If a hardware random number generator were used, `monte` would have nearly 50% robust instructions and would show an energy reduction of up to 17%.

Savings with Medium Energy Settings. With a medium portion of the chip mode switched (Figure 5.1(center)) the Class 1 benchmarks with a high fraction of robust instructions still show modest energy benefits (6%-17%) from deferred execution. However, the gains from low reliability mode are not enough to offset the overheads of switching for the other classes.

Savings with Mild Energy Settings. In the mild energy setting (Figure 5.1(right)), we see no energy savings at all. We see that by only applying a low reliability mode to functional units, there are meager energy savings to be had even in the ideal case, much less in a realizable implementation.

Remaining Overheads. However, even in the mild design point there is still potential for achieving a 2%-7% energy reduction. There are two overheads incurred by our deferred execution implementation inhibiting achieving further energy reductions: (1) the remaining switching overhead (E_{sw}) and (2) additional hardware support overhead (E_Q).

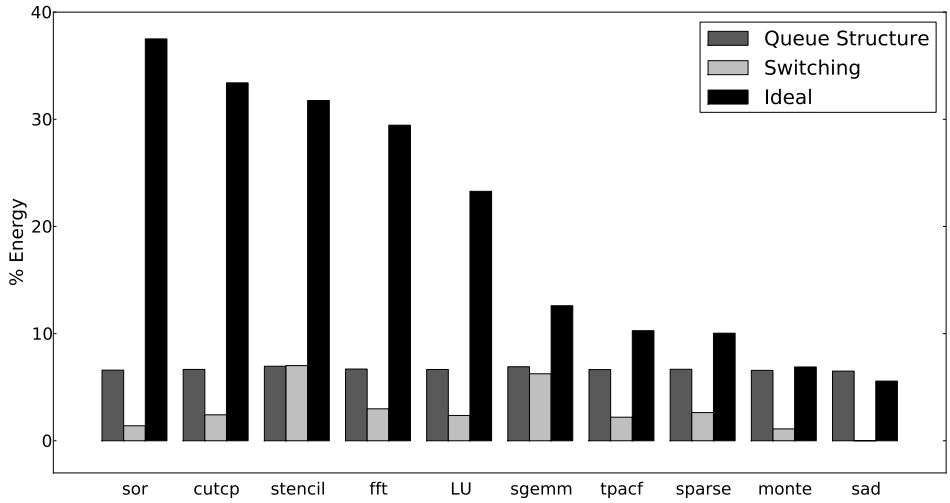

Figure 5.2: **Breakdown of Energy Overheads.** Two overheads remain using the deferred execution model: (1) the switching overhead due to a finite queue capacity and (2) the overhead of hardware support.

These two remaining overheads are compared to the ideal energy savings for each benchmark in Figure 5.2. The hardware queue itself represents a relatively constant 6.5%-7% energy overhead for all the benchmarks, while the energy consumed by the remaining switching overhead is on average 2.84%, and varies between 0% and 6.25%.

For most benchmarks the remaining switches are a result of the significant amount of data addresses that must be transferred from the delicate partition to the robust partition, which causes the queue to frequently fill up thereby triggering switches. Further mechanisms such as various forms of compression may further alleviate this overhead, but we leave that to future work. While a deferred execution model can significantly limit the impact of switching overhead, the queue size versus power tradeoff limits the reduction of switches by limiting the size of the queue structure itself.

Switching Overhead after Optimizations. Figure 5.3 summarizes our evaluation of how each of the aggregation optimizations impacts the average switching period $(N_{Del} + N_{Rob})/N_{Sw}$, that is the average number of instructions before a mode switch is triggered. In the figure, the y-axis is the average switching period for the corresponding benchmark shown on the x-axis. For each benchmark there are four bars representing the execution using no aggregation (i.e., the naive architecture), with static aggregation, deferred execution with basic hardware support, and deferred execution us-

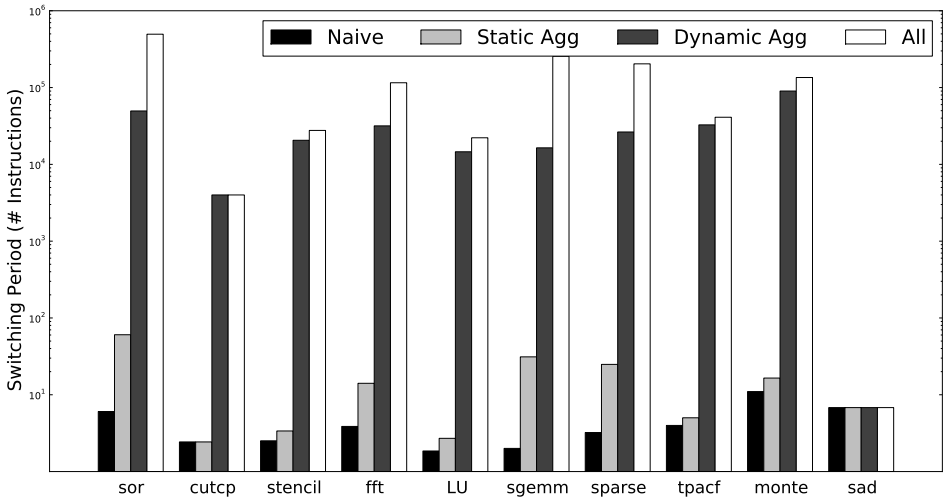

Figure 5.3: **Optimizations' Impact on Switching.** A larger switching period results in less total switching overhead. For most benchmarks dynamic aggregation provides several orders of magnitude improvement.

ing all optimizations. These results assume that a fixed buffer size of 64KB is used.

We observe that static aggregation increases the low switching period arising from finely interleaved robust and delicate instructions by up to 10% in several benchmarks (e.g., `sor` and `fft`), yet does not provide this increase for all benchmarks. With deferred execution, however, the switching period is drastically increased by 2-3 orders of magnitude for most benchmarks.

Therefore, static aggregation clearly does not sufficiently reduce the switching overhead to warrant using this technique alone, instead the deferred execution provides the most dramatic increase in the switching period, so that energy benefits might be yielded.

Sensitivity of Energy Savings to Switching Overhead and Queue Size. Figure 5.4 shows the sensitivity of the average energy benefits to two critical design parameters, the switching overhead and the queue size. Note that a bigger queue means fewer switches (i.e., lower N_{Sw}) but a larger energy cost (i.e., higher E_Q). With typical switching overheads found in the error-resilient architectures [7, 6] we find the optimal queue size needed is around 16KB to 64KB. This is comparable with the cache sizes found within current processor designs, meaning the area/power overhead from using either a dedicated hardware queue or shared cache for the queue would be small. Further, queues of this size may leave flexibility in the design of a

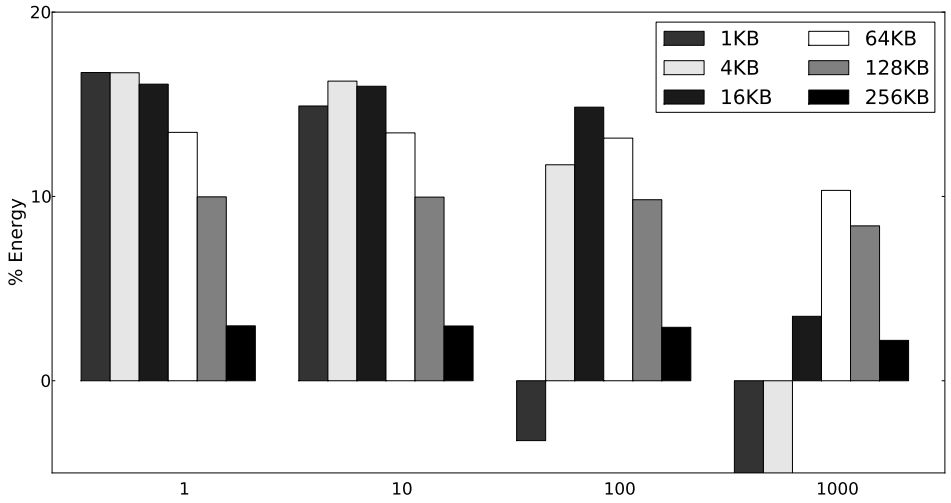

Figure 5.4: **Energy Savings vs. Buffer Size and Switching Overhead.** We determine optimal queue size based on switching overhead (E_{sw}); for $E_{sw} = 1000$ a 64KB queue provides the best energy reductions.

shared cache that allows for the queue size to be dynamically adjusted according the application needs. At a switching overhead of $E_{sw} = 1000$ a 64KB queue is optimal while small queues (<16KB) cannot reduce energy. For switching overheads of $E_{sw} = 1$, 10, or 100, energy benefits from exploiting robust instructions via deferred execution are seen at all queue sizes less than 512KB, with 2KB, 8KB, and 16KB being optimal, respectively. At any larger switching overhead (≥ 10000) no queue size produces energy benefits, but the optimal queue size becomes larger (e.g., 256KB at $E_{sw} = 10000$). Using these results we can optimize the queue structure size given the amount of switching overhead our design would incur. For example with the switching overhead of 1000 a 64KB queue size is optimal.

Chapter 6

Related Work

Next we discuss related work on *opportunitistic energy reduction* via *purposefully* unreliable hardware or software.

Purely Hardware Approaches. Recent research in digital signal processing and circuit design have explored the possibility of exploiting timing and logic failures to improve the energy-efficiency of hardware. Algorithmic noise-tolerance [15] proposes a scheme for voltage over-scaling digital filters while mitigating timing errors via error control. In contrast, Kulkarni et al. [16] explore the use of logic errors, instead of timing errors, to produce more energy-efficient circuits: they propose a multiplier constructed from logically incorrect building blocks that allow for design-time tradeoffs between correctness and energy savings. Similar techniques have been proposed in the context of floating-point units [17], integer logic units [18], and value memoization [19]. At the microarchitectural level, researchers have developed techniques to improve energy efficiency without introducing software-visible error even in the face of timing failures: the Razor architecture [20] employs error-detection circuits to permit aggressive energy optimizations such as power gating, voltage and frequency scaling, and sub-threshold operation with minimal guardband. A common theme among hardware-focused solutions is that these techniques are transparent to software, which limit their ability to exploit application-specific opportunities for energy savings. A key focus of our work is to demonstrate that our deferred execution model, by exposing errors to software, is capable of exploiting application-level error resilience to achieve significant energy benefits.

Purely Software-Based Approaches. Prominent among software approaches to opportunistic energy reduction is the notion of acceptability oriented computation [21, 22]. Through code transformations that approximate the original program semantics, techniques such as loop perforation [21] can extract energy savings at the cost of degraded quality of service. These

techniques apply in exactly those cases in which a high proportion of instructions are robust by our definition. In a related vein, Green [2] proposes a software programming model and runtime system for dynamically tuning approximation decisions via a feedback loop between programmer-specified approximation, calibration, and quality of service monitoring routines. The focus of their work is a software-only mechanism for monitoring quality of service, while our work focuses on a related, but different concern: given a program that we know is error resilient, how can we efficiently extract its potential for energy savings with only lightweight modifications?

Error-Resilient Architectures. Closest in spirit to our work are approaches that employ co-designed architectural and software capabilities for exploiting application level error resilience. Liu et al. [23] propose software modulation of DRAM refresh rates to selectively lower data integrity of global and heap data structures. De Kruijf, et al. [24] present Relax, an architectural framework with ISA extensions for demarcating regions of code for executing under relaxed reliability. Narayanan et al. [25] propose allowing software to selectively use voltage overscaled functional units as well as switching reliability modes at a much coarser granularity based on the power and performance requirements of applications. The ERSA architecture [4] employs a multicore design in which unreliable cores cooperate with a reliable core at a coarse granularity to preserve quality of service for a class of parallel application. Unlike previous work, we focus on automatic techniques for extracting coarse-grained regions for reliability mode switching.

The EnerJ language [3] and Truffles architecture [5] present a high level language and architecture for software-exposed hardware unreliability: the EnerJ type system allows type-directed approximation while disallowing flow from approximate computation into values that must be precise. The Truffles architecture then serves as an execution substrate for such a language: the machine language is extended with approximate variants of arithmetic and logical instructions, and hardware structures are duplicated or modified to allow very fine-grained switching between reliability modes on the order of a single cycle. Although similar in goal to our system, to support such fine-grained interleaving of approximate operations, Truffles requires pervasive and complex circuit modifications such as voltage-translation circuitry, duplication of functional units and pipeline registers, and a dual-voltage cache

with voltage modulation at the cache-line granularity. In contrast, we present a lightweight design with modular changes (an additional queue and context unit) along with static and dynamic optimizations that allow us to achieve energy benefits at coarser granularity.

Reliability Software. Other research focuses on improving reliability for a given performance target [14]. Their work identifies a rigorous methodology for quantifying the error tolerance of static instructions. Using these quantities various code scheduling optimizations (e.g., loop unrolling) were proposed to improve reliability by reducing the number of highly vulnerable instructions. However, this work focuses on improving reliability through software-only optimizations rather than reducing power using both software and hardware techniques.

Chapter 7

Conclusion

Many programs contain a latent potential for energy savings by executing their robust instructions with relaxed reliability. However, we have identified a key program characteristic, *instruction interleaving*, that significantly impacts the ability to realize this potential, due to the high cost of switching reliability modes. Indeed, a naive approach that employs eager switching has dramatic overheads (over 300x on average) that completely negate any energy benefits. In response, we propose a *deferred execution model* in which we defer and aggregate robust instructions to be executed in batch. With this approach, we can reduce the number of mode switches by several orders of magnitude. Although the introduction of a hardware queue introduces new data transfer overheads, we propose novel low-overhead hardware support for minimizing this overhead, allowing us to demonstrate up to 29% energy savings.

References

[1] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010, pp. 25–34.

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010, pp. 198–209.

[3] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011, pp. 164–174.

[4] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010, pp. 1560–1565.

[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012, pp. 301–312.

[6] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *HPCA*, 2008, pp. 123–134.

[7] T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling," in *ISLPED*, 2000, pp. 9–14.

[8] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[9] R. Pozo and B. Miller, "SciMark2," Aug. 2012. [Online]. Available: http://math.nist.gov/scimark2/index.html

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[11] N. Verma, "Ultra-low-power SRAM design in high variability advanced CMOS," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 2009.

[12] M. Qazi, M. Sinangil, and A. Chandrakasan, "Challenges and directions for low-voltage SRAM," *Design Test of Computers, IEEE*, vol. 28, no. 1, pp. 32–43, Jan.-Feb. 2011.

[13] D. Jeon, Y. Kim, I. Lee, Z. Zhang, D. Blaauw, and D. Sylvester, "A 470mv 2.7mw feature extraction accelerator for micro autonomous vehicle navigation in 28nm cmos," in *ISSCC*, in press, 2013.

[14] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: Embedded code generation aiming at reliability," in *CODES+ISSS*, 2011, pp. 237–246.

[15] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999, pp. 30–35.

[16] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSID*, 2011.

[17] J. Tong, D. Nagle, and R. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *VLSI Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 273–286, 2000.

[18] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications," *Trans. Cir. Sys. Part I*, vol. 57, no. 4, pp. 850–862, Apr. 2010.

[19] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, pp. 922–927, July 2005.

[20] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO*, 2003.

[21] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ESEC/FSE*, 2011, pp. 124–134.

[22] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *POPL*, 2012, pp. 441–454.

[23] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *ASPLOS*, 2011, pp. 213–224.

[24] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010, pp. 497–508.

[25] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010, pp. 335–338.