Relax: An Architectural Framework for Software Recovery of Hardware Faults

Marc de Kruijf, Shuou Nomura, Karthikeyan Sankaralingam
Vertical Research Group
University of Wisconsin – Madison
{dekruijf, nomura, karu}@cs.wisc.edu

ABSTRACT

As technology scales ever further, device unreliability is creating excessive complexity for hardware to maintain the illusion of perfect operation. In this paper, we consider whether exposing hardware fault information to software and allowing software to control fault recovery simplifies hardware design and helps technology scaling.

The combination of emerging applications and emerging many-core architectures makes software recovery a viable alternative to hardware-based fault recovery. Emerging applications tend to have *few I/O and memory side-effects*, which limits the amount of information that needs checkpointing, and they allow *discarding individual sub-computations* with small qualitative impact. Software recovery can harness these properties in ways that hardware recovery cannot.

We describe Relax, an architectural framework for software recovery of hardware faults. Relax includes three core components: (1) an ISA extension that allows software to mark regions of code for software recovery, (2) a hardware organization that simplifies reliability considerations and provides energy efficiency with hardware recovery support removed, and (3) software support for compilers and programmers to utilize the Relax ISA. Applying Relax to counter the effects of process variation, our results show a 20% energy efficiency improvement for PARSEC applications with only minimal source code changes and simpler hardware.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/Software Interfaces*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault Tolerance*

General Terms

Design, Performance, Reliability

1. INTRODUCTION

As CMOS technology scales, individual transistor components will soon consist of only a handful of atoms. At these sizes, transistors are extremely difficult to control in terms of their individual power and performance characteristics, their susceptibility to soft errors caused by particle strikes, the rate at which their performance degrades over time, and their manufacturability – concerns commonly referred to as *variability*, *soft errors*, *wear-out*, and *yield*, respectively. Already, the illusion that hardware is perfect is becoming hard to maintain at the VLSI circuit design, CAD, and manufacturing layers. Moreover, opportunities for energy efficiency are lost due to the conservative voltage and frequency assumptions necessary to overcome unpredictability.

This trend towards increasingly unreliable hardware has led to an abundance of work on hardware fault detection [21, 25, 27, 33, 36] and recovery [3, 8, 32, 38]. Additionally, researchers have explored architectural pruning [26] and timing speculation [12, 14, 15] as ways to mitigate chip design and manufacturing constraints. However, in all cases these proposals have focused on conventional applications running on conventional architectures, with a typical separation of hardware and software concerns.

In this paper, we observe two complementary trends in emerging applications and architectures that favor a new overall architectural vision: hardware faults recovered in software. Below, we explain these trends, articulate the challenges in designing an architecture with software recovery, and finally describe our proposed framework, Relax.

Emerging applications – applications that continue to drive increases in chip performance – include computer vision, data mining, search, media processing, and data-intensive scientific applications. Many of these applications have two distinct characteristics that make them interesting from a reliability perspective. First, and a key observation unique to this work, is that many have *few memory side-effects* at the core of their computation. In particular, state-modifying I/O operations are rare and memory operations are primarily loads, because the compute regions of these applications perform reductions over large amounts of data. Second, for many emerging applications, a perfect answer is not attainable due to the inherent computational complexity of the problem and/or noisy input data. Therefore, they employ approximation techniques to maximize the qualitative "usefulness" of their output. This suggests that these applications might be *error tolerant*, which has been observed in prior work as well [6, 11, 22, 23, 42]. In this paper, we specifically explore the phenomenon that the application can *discard* computations in the event of an error.

The concurrent architecture trend is that massively multicore architectures are emerging to meet the computational

ISCA'10, June 19–23, 2010, Saint-Malo, France.

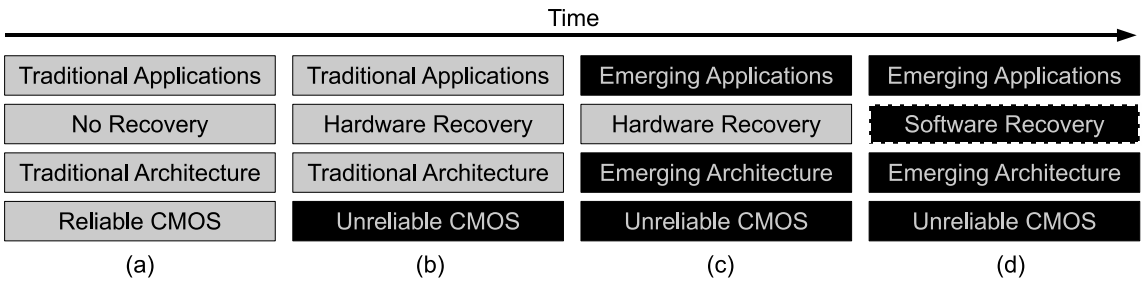

Figure 1: The evolution of hardware, architecture, and applications in the context of Relax.

demands of emerging applications [13, 16, 19]. These architectures often employ simple, in-order cores to maximize throughput and energy efficiency with little or no support for speculative execution or buffering. Hence, the paradigm that hardware misspeculation-recovery mechanisms can be repurposed for error recovery does not apply for these architectures. The valuable chip real estate that would otherwise be devoted to hardware recovery resources could be better spent elsewhere if software recovery were efficient.

Overall, the combination of limited side-effects and error tolerance that exists in large portions of emerging applications renders hardware recovery inflexible, unnecessarily conservative, and too expensive for emerging many-core architectures. Figure 1 shows the evolutionary path to software recovery considering these trends in hardware, architecture, and applications. Historically, traditional applications running on traditional superscalar processor architectures built with perfect CMOS devices required no recovery (Figure 1(a)). Even with imperfect CMOS, these applications still work best utilizing hardware recovery when running on traditional processor architectures (Figure 1(b)). However, with emerging applications running on emerging many-core architectures, hardware recovery introduces the inefficiencies we have described (Figure 1(c)). In the future, while hardware substrates will be unreliable, we require mechanisms that provide flexibility to software and keep the architecture simple. An architecture that exposes hardware errors to allow software recovery enables synergy between applications and architectures as shown in Figure 1(d).

The design of a system architecture that allows such software recovery of hardware faults involves many important questions and challenges. The first and most obvious question is whether changes to the ISA are necessary. To answer this question, we refer to prior studies that show application tolerance to arbitrary instruction-level errors is very poor [6, 11, 23, 22, 42]. Operations relating to control flow and memory accesses are failure prone and constitute a large percentage of application operations. For an architecture to allow reasonably fine-grained software recovery without ISA changes, it would be necessary for the hardware to somehow distinguish these "critical" operations from the "non-critical" operations as it executes code. To date, no one has been able to propose an efficient way to do this. Hence, ISA support appears necessary.

The next logical question concerns what form ISA support should take. Software recovery of hardware faults has been proposed before in the context of software detection, using compiler-automated triple-modular redundancy (TMR) [8]. TMR makes sense when the overhead of detection is already very high, as is the case with comprehensive software detection. However, it is expensive and does not allow the application to exploit error tolerance. A more efficient solution that allows an application to choose its own form of recovery is closer to ideal.

Yet still more questions follow: How might the application writer express software recovery in the application? How can applications be designed to behave predictably when errors occur non-deterministically? Are there ways in which the software development process can be automated or assisted? What should be the hardware organization – should all cores have no recovery support, or just some cores? Are there special considerations for the hardware microarchitecture?

In this paper, we propose a holistic architectural framework, called Relax, that provides specific answers to each of these questions. We divide Relax into three core components: (1) an ISA extension, (2) hardware support to implement the Relax ISA, and (3) software support for applications to use the Relax ISA. We discuss each component in a separate section:

- **ISA extension:** In Section 2, we describe the Relax ISA extension, which enables software to register a fault handler for a region of code. The extension allows applications to encode behavior similar to the *try/catch* behavior found in modern programming languages. The ISA behavior is intuitive to programmers, and the compiler and hardware combine to make guarantees about the state of the program as the region is executed. We also provide a rigorous definition of the ISA's semantics.

- **Hardware support:** We cover the hardware support for Relax in Section 3. The Relax ISA's semantics allow hardware design simplification and provide energy efficiency by relaxing the reliability constraints of the hardware. We describe support for fault detection and discuss hardware organizations that support Relax. We show that mechanisms such as aggressive voltage scaling, frequency overclocking, and turning off recovery mechanisms provide adaptive support for Relax. We also consider statically heterogeneous architectures, where cores are constructed with different reliability guarantees at design time.

- **Software support:** In Section 4, we develop a C/C++ language-level recovery construct to expose the Relax ISA extension to developers. We propose two key ideas: *relax blocks* to mark regions that may experience a hardware fault, and optional *recover blocks* to specify recovery code if a fault occurs. Our results indicate promise for alternative forms of application support as well, such as automated support through compiler static analysis or profile-guided compilation.

To support Relax, we develop performance models to guide the development of "relaxed" applications. The models, discussed in Section 5, determine the efficiency of Relax based on application and architecture characteristics and can be used to compute the achievable efficiency improvements for a given application, recovery behavior, and architecture combination. We evaluate Relax in Sections 6 and 7, where we apply our language construct and Relax compiler to real applications, and simulate how Relax enables energy efficiency gains using process variation as a case study. We discuss directions for future work in Section 8, related work in Section 9, and finally we conclude in Section 10.

2. ISA SUPPORT

In this section, we discuss the ISA component of the Relax framework. In Section 2.1, we describe the Relax ISA extension and briefly introduce our language-level construct, which we use to illustrate how high-level recovery behavior is mapped onto the ISA. In Section 2.2 we describe the ISA semantics in detail.

2.1 ISA & Compiler Support

We sketch a simple C function example to motivate and explain software recovery, and use this example to introduce Relax's ISA extension. Code Listing 1 shows a simple C function and how it is augmented with Relax support and compiled to a sequence of instructions. Listing 1(a) shows the simple summation function and Listing 1(b) shows this function augmented to use Relax. The function uses our *relax/recover* construct, which is analogous to the *try/catch* construct of high-level languages that support exceptions. For the purposes of the example, the next paragraph provides a brief overview of the construct. Section 4 gives more details and uses.

In Code Listing 1(b), all code except the return statement is wrapped in a *relax block*. Code inside a relax block is susceptible to failure, where a hardware fault detected inside the block constitutes failure. The optional variable `rate` specifies a relax block's probability of failure. Without it, the hardware dictates this probability independent of the application. In some situations, this variable is important to make reasonable guarantees about the quality of an application's output. If a failure occurs, control transfers to the *recover block*. In this case, the recover block contains a retry statement, which causes re-execution of the relax block.

To support this behavior through the ISA, Code Listing 1(c) shows the assembly code for this function with the Relax additions highlighted. For readability, we use symbolic register names rather than numbered registers. A single instruction (`rlx`) communicates the start and end of relax blocks to the hardware. When used to enter a relax block, the `rlx` instruction optionally reads a general purpose register containing the desired failure rate, as well as the offset of the PC address to the recovery block, to which the hardware automatically transfers control on failure. The same instruction with a PC offset of 0 signals the end of the relax block. Within the relax block, the execution semantics of the hardware are relaxed. A rigorous definition of what this means follows in Section 2.2.

Compiler support for Relax is relatively straightforward. The compiler sets up the recovery block and adds compensating code to save or recover state if necessary. In the case of the example function, the function has no side-effects and therefore has no state, beyond its input state, that needs to be restored in the event of a failure. If a failure occurs inside the function, it is sufficient to simply jump back to the beginning of the function, as Code Listing 1(c) demonstrates, with the guarantee that the input registers have not been overwritten. The compiler transparently enforces this guarantee simply by knowing that such a control path exists, thereby effectively implementing a software checkpoint. The checkpoint is extremely lightweight: the compiler only saves state that is strictly required. In this case, the two inputs, `list` and `len`, must either be saved to the program stack or must occupy available registers. Five physical registers are needed to store all the live variables in this function. If five are available, Relax adds no software overhead.

2.2 ISA Semantics

Relax allows instructions to commit potentially erroneous state, while the compiler ensures that this state is either discarded or overwritten after the fault is discovered and recovery is initiated. For the compiler to ensure recovery from the fault, the resulting error must be a *Locally Correctable Error* (LCE), as defined by Sridharan et al. [39]. Hence, the error must be spatially and temporally contained, which forces the following hardware constraints:

1. Errors must be spatially contained to the target resources of a relax block's execution. In other words, an instruction must not commit corrupted state to a register or memory location not written to by other instructions in the relax block. For stores, this means that a store must not commit if its destination address is corrupt, or if the store is reached through erroneous control flow. A simple (but high overhead) way to handle this is to stall on the error detection logic prior to committing a store. For other instructions that write only to registers, a tight coupling between the detection logic of the destination register datapath and the instruction commit logic enables rapid resolution of writes to incorrect destination registers.
2. The contents of memory locations must not spontaneously change, e.g. due to a particle strike. Relax depends on traditional mechanisms such as ECC to protect memories, caches, and registers from soft errors. Other errors that cannot be temporally contained to the scope of a relax block, such as most faults in the cache coherence or cache writeback logic, are also not recoverable by Relax.
3. Arbitrary control flow is not allowed. Control flow must follow the program's static control flow edges. Note that faulty control *decisions* are still acceptable since the static control flow is not violated.
4. Hardware exceptions must not trigger until hardware detection ensures that the exception is not the result of an undetected hardware fault.
5. Specifically under retry behavior (as in the example of Code Listing 1), an instruction may not store to a volatile address: on re-execution, the store might write to a different address and the initial store is then an irreversible data corruption. Atomic read-modify-write operations, such as an atomic increment, are also problematic to handle under retry behavior without violating the atomicity constraint. For this reason, relax blocks using retry may not currently contain any atomic read-modify-write operations.

Code Listing 1 A simple summation function (a) modified to use Relax (b) and the assembly output produced by the compiler (c). For (c), the Relax additions are in bold. The `RECOVER` label can be folded away but is included for clarity.

```
int sum(int *list, int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += list[i];
  }
  return sum;
}
```

(a)

```
int sum(int *list, int len) {
  relax (rate) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
      sum += list[i];
    }
  } recover { retry; }
  return sum;
}
```

(b)

```
ENTRY:
  rlx ${rate}, RECOVER  # Relax on
  mv  0 -> $sum
  ble $len, 0, EXIT
LOOP_PREHEADER:
  mv  0 -> $i
LOOP:
  sll $i, 2 -> $tmp
  ld  [$list + $tmp] -> $tmp
  add $sum, $tmp -> $sum
  add $i, 1 -> $i
  blt $i, $len, LOOP
EXIT:
  rlx 0   # Relax off
  ret $sum
RECOVER:  # Relax automatically off
  jmp ENTRY
```

(c)

```
    RECOVER:
√   rlx ${1/rate}, RECOVER
√   mv  0 -> $sum
√   ble $len, 0, EXIT
X   mv  0 -> $i
    sll $i, 2 -> $tmp
?   ld  [$list + $tmp] -> $tmp
```

Figure 2: An example of Relax's execution behavior.

Execution may leave a relax block once the hardware detection guarantees error-free execution. In the event of an error, the hardware must trigger recovery at some point before execution leaves the relax block.

An example that illustrates Relax's ISA semantics in action is shown in Figure 2. It uses the instruction stream from Code Listing 1(c). The `rlx`, `mv`, and `ble` instructions all complete and commit successfully but a fault occurs executing the second `mv` that is initially undetected and so the instruction commits as normal. Next, the result of the `sll` instruction is pipeline bypassed to the `ld` instruction. When the `ld` executes it triggers a page fault exception due to its corrupted input address. Before the exception is handled, the hardware waits for the detection to catch up. The fault from the `mv` is detected and execution jumps back to the `RECOVER` label.

3. HARDWARE SUPPORT

In this section, we present the hardware component of Relax. The Relax ISA's main hardware benefits are design simplification and energy efficiency, while the key hardware requirement is fault detection. We first discuss Relax's hardware benefits, followed by the hardware detection support. We conclude with a description of the overall hardware organization.

3.1 Hardware Simplification

Relax provides sevaral hardware benefits. First, the hardware need not provide support for buffering, checkpointing, or rollback for software-recoverable errors. Second, complicated techniques to combat parameter variations and wearout, such as fine-grained body biasing [40], are less useful under Relax because, by design, variations are more tolerable. Finally, Relax reduces hardware design complexity because design margins to account for silicon uncertainity can be relaxed. This also potentially improves energy efficiency, as it allows hardware to be designed for correct and efficient operation under common case conditions, but with possible failures under dynamically worst case conditions. The overall result is hardware that is error-prone, but is easier to design and potentially more energy efficient. In Section 7, we consider timing faults from process variations and show how Relax provides energy efficiency and design complexity benefits.

3.2 Hardware Detection

Relax requires support for low-latency fault detection in hardware. Two viable alternatives are Argus [25] and redundant multi-threading (RMT) [27]. Argus provides comprehensive error detection specifically targeted at simple cores, and RMT runs two copies of a program on separate hardware threads and compares their outputs to detect faults. In addition, Razor [12] describes support for adaptive failure rate monitoring for timing faults. Relax requires a similar mechanism to ensure the fault rate remains stable if the `rlx` instruction's target fault rate input is specified.

3.3 Hardware Organization

While hardware that implements Relax everywhere and has no recovery support at all is the ideal, it is disruptively different from existing hardware and requires complete software support. Other configurations that partially implement Relax can be incrementally built into existing hardware organizations. In this section, we consider in detail three such organizations with both relaxed hardware

Relaxed Hardware Implementation	Recover Cost	Transition Cost
Fine-grained tasks	5	5
DVFS	5	50
Architectural core salvaging	50	0

Table 1: Parameters for three alternative relaxed hardware designs.

and normal hardware, where relax blocks execute on relaxed hardware and other code executes on normal hardware.

Whether hardware is relaxed or not can be configured either statically or dynamically. In the static case, two types of cores are used: relaxed cores and normal cores. Relax blocks are off-loaded to relaxed cores and other code executes on normal cores. The relaxed cores can use less design guardband and do not need any hardware recovery mechanism. In the dynamic case, circuit techniques like voltage scaling or frequency over-clocking are used to execute relaxed blocks with improved overall efficiency and/or hardware recovery support can be adaptively disabled.

The type of hardware organization affects the performance of Relax. In particular, two costs dictated by the hardware are important: (1) the cost in cycles to detect and initiate recovery, and (2) the cost in cycles to transition into and out of relax blocks. Table 1 gives estimates for these two costs for three different hardware alternatives we examine.

The first alternative is a statically configured architecture with support for fine-grained parallelism, where relax blocks are enqueued on a neighboring, unreliable core with low latency (e.g. Carbon [19]). The cost to recover is the cost of a pipeline flush, approximated at 5 cycles for a simple in-order core, and the cost to transition is the time to enqueue a task, which we estimate at 5 cycles. The second alternative is a dynamically configured architecture that uses dynamic frequency and voltage scaling (DVFS) to enter and exit relax blocks (e.g. Paceline [14]). The cost to recover is again just the cost of a pipeline flush, and we approximate the cost of DVFS at 50 cycles, which the work of Kim et al. suggests is reasonable for on-chip DVFS [17]. Finally, we consider an organization where hardware recovery is adaptively disabled and a thread swap occurs with a neighboring core in the event of a fault (e.g. Architectural Core Salvaging [31]). We assume the cost of a thread swap to recover is 50 cycles, with no cost to transition. We revisit the values in Table 1 when we discuss performance models in Section 5.

4. SOFTWARE SUPPORT

In this section, we use the recovery construct introduced in Section 2.1 to demonstrate how Relax enables the implementation of flexible and efficient recovery policies in software through a series of example use cases. Our use cases derive from the code shown in Code Listing 2, adapted from the `x264` video encoding application. The listing shows a C function returning the *sum of absolute differences* over the array inputs `left` and `right`. It provides an example of a computation that is well suited for software-level recovery.

Although this example is taken from `x264`, many modern, computationally-intensive applications employ computation such as this, i.e. reduction, at the core of their execution. `x264` uses a two-dimensional version of this function to search for a predicted frame macroblock's most similar

Code Listing 2 The *sum of absolute differences* code example that is the basis for all use cases.

```
int sad(int *left, int *right, int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i)
    sum += abs(left[i] - right[i]);
  return sum;
}
```

reference frame macroblock. The function measures similarity by performing a pixel by pixel comparison over two macroblocks. A high similarity presents redundancy that can be exploited to minimize the amount of information encoded. The overall process is called motion estimation, which allows for better data compression. The four use cases we explore each perform a different type of recovery over this function. We consider two high-level recovery behaviors: retry (Re) and discard (Di), furthermore distinguished by their granularities: coarse-grained (Co) and fine-grained (Fi). Table 2 illustrates the resulting taxonomy.

Use Case 1: Coarse-Grained Retry (CoRe). Relax and recover blocks can be used to implement coarse-grained retry (CoRe) as shown in the upper-left quadrant of Table 2. This case is the same as shown for the example presented in Section 2.1. Just like the `sum` function, the `sad` function has no memory side-effects and therefore execution can simply jump back to the beginning of the function if a fault occurs, provided the inputs are still available. The Relax compiler performs a control flow analysis over the relax block, sets up the recovery code, and adds compensating code to save or recover state if necessary.

Use Case 2: Coarse-Grained Discard (CoDi). Three difficulties with CoRe are that it (1) potentially requires saving and restoring software state, (2) requires a retry mechanism that can deflect recurring failures, and (3) can hurt performance predictability. For error-tolerant applications, particularly those with real-time constraints, a potentially better alternative is to simply abort the function and return a value that indicates the function output should be disregarded. The code in the upper-right quadrant of Table 2 explores this alternative. In the case of `x264`, returning a maximum integer value effectively tells the application to disregard this macroblock pair and continue looking. Similar to CoRe, this use case operates at a coarse granularity.

Use Case 3: Fine-Grained Retry (FiRe). Another alternative to CoRe is to retry at a finer granularity to minimize the amount of wasted work on failure. This can be done simply by moving the relax block into the loop as shown in the lower-left quadrant of Table 2. In this case, each individual accumulation is retried on failure. Since the last instruction of the relax block is the accumulation onto `sum`, the old value of `sum` can be immediately overwritten as the block terminates.

Use Case 4: Fine-Grained Discard (FiDi). For functions that allow approximate output, individual accumulation values can be discarded as shown in the lower-right quadrant of Table 2. Note that there is only a single relax block and no recover block. The resulting behavior is as if there was a recover block that was empty (omitting it en-

	Retry	Discard

Coarse-grained	`int sad(int *left, int *right, int len) {`
	`  relax (rate) {`
	`  int sum = 0;`
	`  for (int i = 0; i < len; ++i)`
	`  sum += abs(left[i] - right[i]);`
	`  } recover { retry; }`
	`  return sum;`
	`}`
	Use Case 1 (CoRe)
		`int sad(int *left, int *right, int len) {`
		`  relax (rate) {`
		`  int sum = 0;`
		`  for (int i = 0; i < len; ++i)`
		`  sum += abs(left[i] - right[i]);`
		`  } recover { return INT_MAX; }`
		`  return sum;`
		`}`
		Use Case 2 (CoDi)

Fine-grained	`int sad(int *left, int *right, int len) {`
	`  int sum = 0;`
	`  for (int i = 0; i < len; ++i)`
	`  relax (rate) {`
	`  sum += abs(left[i] - right[i]);`
	`  } recover { retry; }`
	`  return sum;`
	`}`
	Use Case 3 (FiRe)
		`int sad(int *left, int *right, int len) {`
		`  int sum = 0;`
		`  for (int i = 0; i < len; ++i)`
		`  relax (rate) {`
		`  sum += abs(left[i] - right[i]);`
		`  }`
		`  return sum;`
		`}`
		Use Case 4 (FiDi)

Table 2: Our four use cases classified by granularity and recovery behavior.

hances readability). Without the recover block, the variable `sum` has two possible values at the end of the relax block: either it has been updated with the new value, or it is unchanged. This achieves the desired behavior: on failure, the accumulation value is discarded.

5. APPLICATION DEVELOPMENT

Relax provides hardware energy efficiency improvements by removing the need for hardware recovery support while still allowing hardware faults to occur. However, there are software overheads associated with Relax. In the case of retry behavior, there is the potential cost of saving and restoring state, and also the overhead of the wasted time spent executing failed relax block executions. In the case of discard behavior, failed relax block executions reduce the application's output quality (e.g. image sharpness). To compensate, the application must be configured at a higher quality setting (e.g. more iterations) to achieve the same output quality. This introduces execution time overhead.

In this section, we develop a set of analytical models to help developers reason about the various efficiency considerations. One of the key outcomes of our models is that, depending on application, recovery behavior (e.g. retry vs. discard), and architecture characteristics, we can determine the specific fault rate that maximizes overall efficiency. The models are extended from the probabilistic models for the performance overhead of backward error recovery developed by De Kruijf et al. [9]. We refer the reader to a technical report for details on how we extended the models for Relax [10]. We focus on energy efficiency and specifically energy-delay product (EDP), although our methodology can be trivially extended to other metrics.

Model for Retry Behavior. Our model for retry behavior uses four primary inputs: *cycles*, the execution time in cycles of a relax block, *recover*, the cost in cycles to initiate recovery, *transition*, the cost of transitions into and out of relax blocks, and *rate*, the per-cycle error rate. Using basic probability theory we define a function to compute the overheads due to re-execution triggered by faults. We combine this with a hardware efficiency function that maps a hardware fault rate to the energy efficiency of the hardware relative to hardware that does not allow any faults. We call this function EDP_{hw}. The functions combine to produce a final function that maps a fault rate to the energy-delay of the whole system. We call this function EDP_{retry}. Solving for the derivative of this equation set to zero yields the fault rate that minimizes overall EDP.

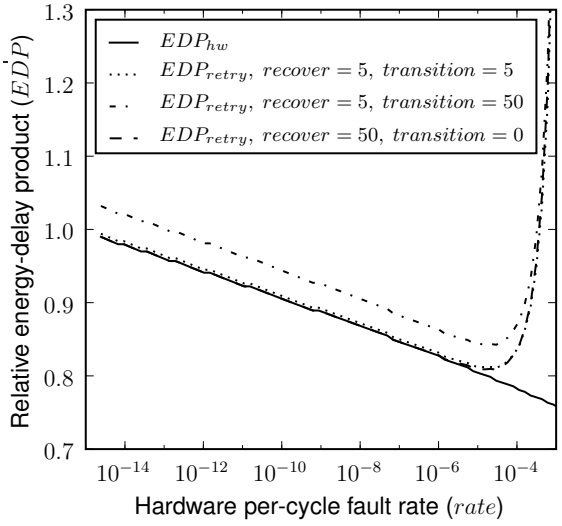

Figure 3: A mapping from fault rate to EDP for different architectural parameters.

For a relax block where *cycles* is roughly 1170, Figure 3 shows a graph evaluating the three hardware organizations from Table 1, using the hardware-specific values for *recover* and *transition* given in columns 2 and 3. We assume there is no overhead to save or restore software state, which we find to be realistic in practice. The solid curve shows a hypothetical EDP_{hw} mapping, which represents the ideal case. The dotted curve considers fine-grained tasks, the dash-dotted curve considers DVFS. and the dashed curve considers architectural core salvaging[1], The figure shows that, for these three hypothetical design points, Relax provides an approxi-

[1]For architectural core salvaging, the thread swap on failure

mately 22.1%, 21.9%, and 18.8% optimal EDP reduction for each, respectively. The optimal fault rates are in the range $1.5e^{-5}$ to $3.0e^{-5}$ faults per cycle.

Model for Discard Behavior. The challenge with discard behavior is that an application's output quality depends on the fault rate. We add a new function that maps a combination of an application's input quality setting and the hardware fault rate to the application's output quality. That is, for a target output quality q_o and an input quality setting q_i, $quality(q_i, rate) = q_o$. Then, the constraint $quality(q_i, rate) = quality(q_{i_{base}}, 0)$ for input qualities q_i and $q_{i_{base}}$ ensures that output quality remains constant with Relax (left-hand side) relative to without Relax (right-hand side). We then, extend the retry models with this function to create $EDP_{discard}$. Our technical report presents a detailed discussion of determining the quality function for a given application and relax block, and give examples on how to apply it to the Relax framework [10].

6. EVALUATION METHODOLOGY

While several phenomena can cause faults to occur in the hardware, we evaluate one specific case here. The scenario we consider is Relax in the context of process variations where the hardware is designed to ignore these variations, resulting in some timing faults. This section discusses our evaluation methodology and the next section presents experimental results.

We implemented language support for relax and recover blocks in C/C++ programs using the LLVM compiler infrastructure [20]. We apply the compiler to applications and simulate them using instruction-level fault injection to estimate the potential energy efficiency gains of Relax when coupled with hardware that runs more efficiently in the presence of faults. In Section 6.1 we describe our methodology for evaluating applications using discard behavior specifically. In Section 6.2 we describe our fault injection methodology and in Section 6.3 we discuss performance metrics. Finally, in Section 6.4 we derive a hardware efficiency function to model the impact of allowing errors due to process variations on hardware energy efficiency.

6.1 Evaluating Discard Behavior

Prior work evaluating application-level error tolerance has employed application-specific quality metrics to assess the degree of output quality deterioration [6, 11, 22, 23]. These studies attempt to hold execution time relatively constant while using the error rate to vary output quality. The difficulty with this approach is that it is fundamentally hard to quantify and evaluate variations in output quality.

We provide a novel solution to this problem by taking the converse approach of holding output quality constant while using the error rate to vary execution time. For each application using discard behavior, we define a function that maps an input quality setting and a fault rate to an output quality, and we use it to adjust the input quality setting as we adjust the fault rate to hold output quality constant. The function is the *quality* function discussed at the end of Section 5, and it allows an apples-to-apples comparison across applications. We provide more detail in our technical report [10].

effectively doubles the fault rate, since the neighboring core must abort as well. This is not modeled.

6.2 Fault Injection

To perform detailed quality analysis for discard behavior as described above, we required a simulation framework that would allow us to run relaxed applications to completion on large, representative input data. To meet this challenge, we developed an LLVM instrumentation pass to perform instruction-level fault injection for rapid simulation. We chose LLVM because its virtual ISA closely matches both the x86 and SPARC V9 instruction sets [2], while instrumenting LLVM bytecode is straightforward and flexible. For fault injection, each LLVM instruction inside a relax block is surrounded by code that probabilistically injects an error into the output of that instruction. Although we inject only single-bit errors, the nature of the error is in practice not relevant since corrupted output is ultimately either discarded or overwritten, and hence is never used.

If an error occurs in the address computation of a store instruction, the store does not commit and execution immediately jumps to the recovery destination. If an error occurs in any other instruction, the instruction commits and execution continues as normal, but a recovery flag is set to indicate that an error occurred. When control reaches the end of the relax block, execution jumps to the recovery destination if the recovery flag is set. This behavior is consistent with the ISA semantics described in Section 2.2.

6.3 Performance Metrics

We use execution cycles to measure performance overheads and energy efficiency improvements. To compute execution cycles we record the number of dynamic LLVM instructions executed (not including instructions added for fault instrumentation) and multiply by the CPL (cycles per LLVM instruction) of the relax block. We similarly divide the per-instruction fault rate by the the CPL to compute the per-cycle fault rate.

The validity of using CPL to produce cycle-accurate performance numbers depends on our ability to assert that CPL does not change when relax blocks are augmented with retry or discard behavior. Below, we explain why the two factors that might affect CPL, *instruction mix* and *memory latency*, are not adversely affected by these behaviors. First, all relax blocks we consider have a largely homogeneous instruction mix. Therefore, partial execution of a relax block has a CPL very close to the overall CPL of the block, and certainly averaged over many millions of executions, the CPL will tend towards the CPL of the whole block. Second, for memory latency, we note that retry behavior will re-execute over data that is already cached, and therefore our measured CPL will be an *overestimate*, while for discard behavior, any early termination will place more weight on up-front loads that bring in potentially uncached compute data, yielding an *underestimate*. We accept the overestimating factor and our results for retry behavior are therefore conservative. For discard behavior, we observe that none of our applications are structured with up-front loads since the relax blocks are in all cases iterating over simple array structures. We assert that the overestimating effect is therefore negligible.

6.4 Hardware Efficiency Model

De Kruijf et al. extend the VARIUS model [35] for process variations to provide estimates for the efficiency of hardware allowing timing faults for the OpenRISC core design [9]. The resulting model outputs the relative energy efficiency

Application Name	Function Name	Function % Exec. Time
barneshut	RecurseForce	>99.9
bodytrack	InsideError	21.9
canneal	swap_cost	89.4
ferret	isOptimal	15.7
kmeans	euclid_dist_2	83.3
raytrace	IntersectTriangleMT	49.4
x264	pixel_sad_16x16	49.2

Table 4: Application functions and percentage of execution time inside each function.

of a given processor design as the error rate is varied. We applied the methodology developed by De Kruijf et al. to develop our hardware efficiency function, which we use in our evaluation to model hardware that runs more efficiently in the presence of timing faults induced by process variations. Details of the function's derivation are in our technical report [10].

7. RESULTS

This section presents results using and evaluating the Relax framework. In Section 7.1 we show evidence for the *error tolerance* phenomenon by identifying applications from the PARSEC benchmark suite that are tolerant to discarded computations. We then show in Section 7.2 the results applying our language constructs to each of these applications. We show that relax block regions account for large portions of application execution times, and that the phenomenon of *limited memory side-effects* allows Relax to work with essentially no software overhead. Finally, using our efficiency mapping driven by process variations, we evaluate energy-delay improvement using Relax in Section 7.3. Overall, our results show that 20% improvement in energy efficiency is common, and the optimal fault rate is highly application dependent, varying by several orders of magnitude.

7.1 Evidence for Error Tolerance

We identified seven applications from the PARSEC benchmark suite [7] employing approximation techniques. However, two applications, fluidanimate and streamcluster, did not have an easily identifiable input quality parameter, which was needed to evaluate discard behavior. Since this was merely an artifact of their implementation, we replaced them with more straightforward alternatives from the same application domain. We replaced fluidanimate with barneshut, a physics application from the Lonestar Benchmark Suite [18], and streamcluster with kmeans, a clustering application from NU-MineBench [29].

Table 3 shows the details for each application. Columns 1-3 show the application name, benchmark suite, and application domain, respectively. Columns 4-5 concern evaluation of discard behavior only, and show the input quality parameter used to configure output quality and the quality evaluator used to evaluate output quality, respectively.

7.2 Application Relaxation

The seven applications were modified to implement the four use cases described in Section 4. For each application, we modified only a single, dominant function to use Relax. More functions exist, but evaluating all of them was beyond the scope of this work. Table 4 identifies each application's function and the percentage of execution time spent inside the function. Percentages were measured using the Google Performance Tools CPU profiler [1] running applications natively on a 2.53 GHz Core 2 Duo processor and include time spent in external library calls.

Six of the seven applications were evaluated for all four use cases FiRe, CoRe, FiRe, and FiDi. Barneshut could only support the two fine-grained use cases FiRe and FiDi. Table 5 shows detailed statistics for each application. Columns 2-5 show the length of each relax block in cycles. Columns 6 and 7 show the percentage of executed LLVM instructions affected by Relax for each use case. Combined with the data from Table 4, we see that for three applications more than 70% of the application is relaxed, for two others roughly 50% is relaxed, and for the last two less than 20% is relaxed. Columns 8 and 9 show the number of C/C++ source code lines modified or added. In all cases, the number of changes is very low. Relax blocks do not appear to obstruct code readability and are in most cases straightforward to implement. Finally, columns 10 through 11 show the number of register spills needed to set up a software checkpoint for retry behavior. The numbers assume an architecture with 16 general purpose integer registers and 16 floating point registers. In all cases, there is no software checkpointing overhead; the functions are side-effect free, and simple enough that there is insufficient register pressure to force additional register spills to save input state. Even with register pressure, the number of extra registers needed is between zero and two.

7.3 Execution Time and Energy Efficiency

Figure 4 shows execution time and energy-delay product (EDP) for each application and use case relative to execution without Relax. The triangles plot fault rate versus execution time and the stars plot fault rate versus EDP. EDP is measured applying our hardware efficiency function to the square of the execution time. The figure also shows the results predicted by our models developed in Section 5; the dotted curves plot the fault rate versus predicted execution time and the solid curves plot the fault rate versus predicted EDP. The x-axis ranges are centered around the predicted optimal fault rate. We model hardware with fine-grained task support, presented earlier as the first entry in Table 1, and hence model the hardware costs to initiate recovery and transition in and out of relax blocks at 5 cycles each.

For retry behavior, the results show that a 20% reduction in EDP is common for CoRe, and that CoRe tends to perform better than FiRe. In some cases, execution time with FiRe is very high, as with kmeans and x264. For these applications the fine-grained relax block size is only 4 cycles, and the 5 cycle cost to transition in and out of the relax block forces high overheads.

For discard behavior, we see two flavors of results: *ideal* and *insensitive*. The graphs are annotated with these labels. In the *ideal* cases, changing the input quality setting and/or injecting errors into the application affects behavior in a way that is very regular and consistent. As a result, the discard behavior results for CoDi and FiDi closely mirror those for CoRe and FiRe. The two differences are that (1) in some cases discard behavior cannot support a fault rate quite as high as retry behavior, and (2) the resulting data are slightly more noisy. However, discard behavior will still be the more desirable alternative when performance predictability is more important than output predictability, as

Application Name	Benchmark Suite	Application Domain	Input Quality Parameter	Quality Evaluator
barneshut (fluidanimate)	Lonestar (PARSEC)	Physics modeling	Distance before approximation	SSD[a] over body positions, relative to maximum quality output
bodytrack	PARSEC	Computer vision	Number of simultaneous body particles	Application-internal likelihood estimate
canneal	PARSEC	Optimization: local search	Number of iterations	Change in output cost, relative to maximum quality output
ferret	PARSEC	Image search	Maximum number of iterations	SSD[a] over top 10 ranking, relative to maximum quality output
kmeans (streamcluster)	NU-MineBench (PARSEC)	Data mining: clustering	Number of iterations	Application-internal validity metric
raytrace	PARSEC	Real-time rendering	Rendering resolution	PSNR of upscaled image, relative to high resolution output
x264	PARSEC	Media encoding	Motion estimation search depth	Encoded output file size relative to maximum quality output

[a]SSD = Sum of squared differences

Table 3: The seven applications modified to use Relax.

Application Name	Relax Block Length in Cycles				Percentage of Function Relaxed		Source Lines Modified		Checkpoint Size (Register Spills)	
	CoRe	CoDi	FiRe	FiDi	CoRe / CoDi	FiRe / FiDi	CoRe / CoDi	FiRe / FiDi	CoRe	FiRe
barneshut	N/A	N/A	98	98	N/A	70.6	N/A	6	N/A	0
bodytrack	775	812	25	25	76.3	47.8	2	2	0	0
canneal	2837	2837	115	115	99.8	62.0	2	8	0	0
ferret	4024	4077	12	11	99.6	72.3	2	4	0	0
kmeans	81	81	4	4	99.5	65.8	2	2	0	0
raytrace[a]	2682	2682	136	136	96.5	67.7	2	6	0	0
x264[a]	1174	1174	4	4	99.9	76.2	2	2	0	0

[a]SSE is emulated for x264 and raytrace

Table 5: Details for each application's function and the various use cases implemented.

might be the case with a real-time ray tracer or an online data clustering algorithm.

The *insensitive* discard behavior cases are bodytrack and x264. For bodytrack, the algorithm effectively only has two outputs: either the tracked body position is close, or it is off because the algorithm has lost a handle on the body position. For the quality settings we used, the algorithm did not lose the body position at fault rates of less than $1e^{-3}$ for CoDi and $2e^{-2}$ for FiDi. Hence, any lower fault rate setting produced effectively equivalent output quality, and, due to the nature of discard behavior, the execution time of the program was shortened by the faults and EDP improved. For x264, the reasons are slightly different. For x264 with the reference input we used, it was very difficult to affect the output quality by adjusting the input quality at all. Even at the lowest setting, with a 40% reduction in execution time, the change in output quality was still only extremely minor. Although our *quality* function was able to capture sufficient variation for FiDi, the range was too narrow for CoDi. Even for FiDi, the function was very noisy. We expect that different data input might lead to different results.

8. FUTURE WORK

The Relax framework presents many interesting directions for future work. We discuss a few of them below.

Architecture Exploration. In this paper, we considered the Relax framework in the context of some hypothetical hardware organizations and their associated parameters. The design of completely relaxed hardware would allow a detailed exploration of the trade-offs involved in implementing the Relax ISA. While we believe the ISA provides hardware simplicity, this design exercise would show how effective it truly is. Other topics for future work include considering phenomena beyond merely process variations, and also extending Relax to encompass faults occurring outside the processor core, which would yield a complete, chip-level recovery framework for hardware faults.

Compiler-Automated Retry Behavior. The key requirement for retry behavior on a region is idempotency, which is guaranteed by the absence of read-modify-write sequences. If, for example, a compiler creates a software checkpoint at the end of each read-modify-write sequence, Relax can be active throughout an entire application's execution. The key read-modify-write sequences to consider are load-store pairs targeting the same global or heap memory location; register spills and refills to and from the program stack are automatically handled by the compiler to preserve idempotency.

Binary Support for Retry Behavior. Applying Relax to static binaries when source code is not available is an-

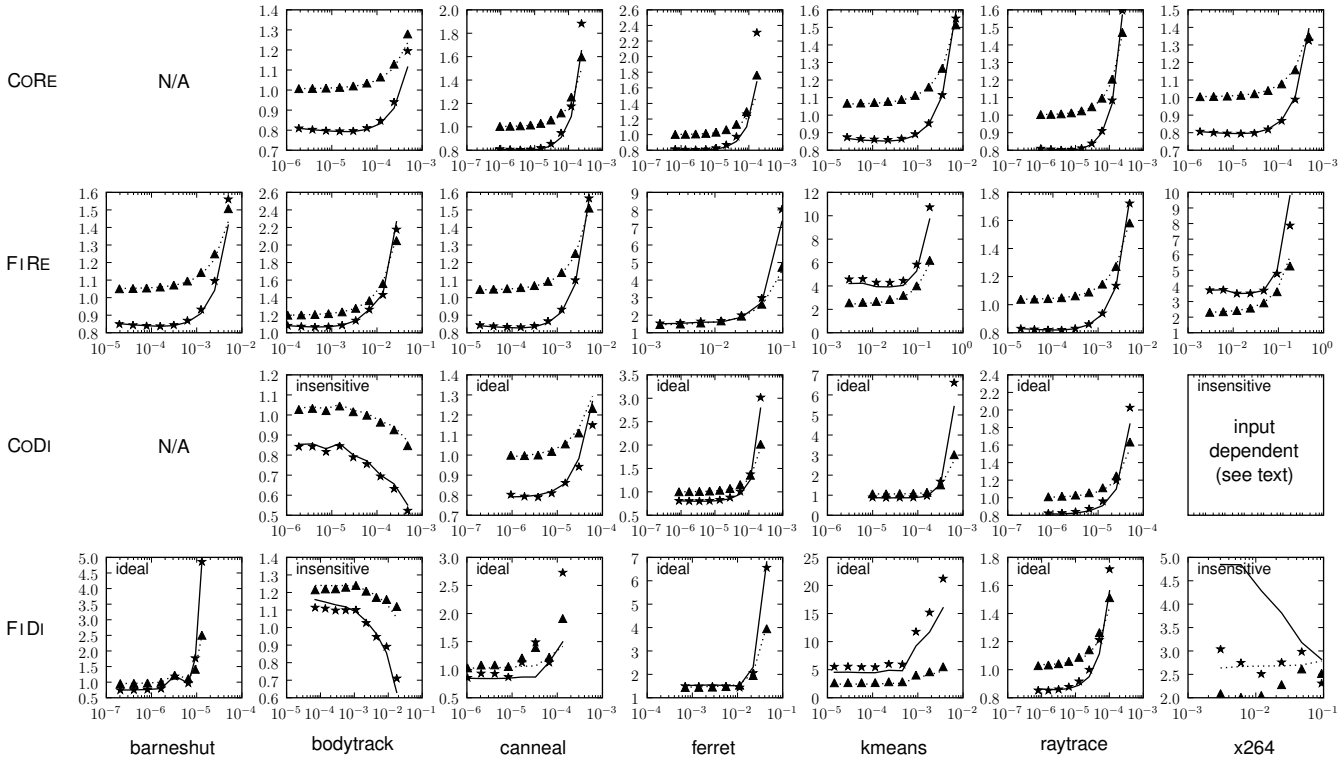

Figure 4: Solid curves plot analytically predicted fault rate (x-axis) versus EDP (y-axis) for each application and use case combination, with empirical data shown using stars. Dashed curves plot fault rate versus execution time only, with empirical data shown using triangles.

other interesting direction for future work. Dynamic binary instrumentation using tools like Pin [24] can be used to dynamically identify good relax block candidates. Static program analysis techniques [5] can also be used to identify idempotent regions in binaries.

Support for Discard Behavior. Discard behavior can be hard to reason about, in part because it exhibits non-determinism. Furthermore, unintentional non-determinism can easily lead to bugs that are very hard to track down. Language support to annotate intentional non-determinism could be used by a compiler or static analysis tool to identify potential bugs in the program. This and other tools to help reason about the efficacy and impact of discard behavior is an important topic for future work.

Nesting Support. Nesting relax blocks inside other relax blocks is a promising idea. The behavior for nested relax blocks should be a straightforward extension of the normal behavior: execution inside relax blocks is relaxed even when nested inside another relax block, and failures cause control to transfer to the end of the innermost relax block. Architecturally, the only requirement to implement this behavior is micro-architectural support for a stack-like structure to store the stack of failure destination addresses, akin to the Return Address Stack (RAS) in modern microprocessors.

9. RELATED WORK

We discuss related work in error recovery, full-system solutions to hardware errors, and application error tolerance.

Error Recovery. Sorin provides a complete treatment of error recovery solutions [37]. He describes two primary approaches to error recovery: *backward error recovery* (BER) and *forward error recovery* (FER). Relax provides BER under retry behavior, and a restricted form of FER under discard behavior. We consider each separately below.

For BER, Relax is distinct from other mechanisms in that it is both software based and has a small sphere of recoverability. Other software approaches have larger spheres of recoverability [30, 41] which comes at a substantial cost to performance. Hardware approaches have both large [32, 38] and small [3, 28] spheres of recoverability. However, hardware checkpoints consume substantial chip resources, and may not even be feasible when dealing with highly error-prone environments, where the checkpointing logic and storage itself cannot be made relatively immune to errors. Relax's fine-grained recovery in software is a good fit for an anticipated future with high fault rate systems running emerging applications that have few memory side-effects and can recover in software with low overhead.

On the FER side, the main competing approach is triple-modular redundancy (TMR). With discard behavior, Relax does not add any redundancy to implement FER, but rather allows the programmer to exploit the redundancy inherent in the application.

Full-System Solutions. Table 6 classifies other full-system proposals for managing error-prone hardware. SWAT [21, 34] uses lightweight symptom- and invariant-based detection techniques combined with heavyweight hardware checkpoints to recover from failure. SWAT optimizes for the

	Recovery	
Detection	Hardware	Software
Hardware	RSDT[4] SWAT [21, 34]	**Relax**
Software	SWAT [21, 34]	Liberty [8, 33]

Table 6: A taxonomy of full-system solutions.

modern-day common case of failure-free execution with a primary focus on reducing detection overhead while latency is not a concern as long as recovery remains possible. Our work is distinct from SWAT in anticipating a future where, for efficiency reasons, failure is much more common, and we shift priorities accordingly. Additionally, Relax is a software recovery framework that utilizes hardware detection, in contrast to SWAT's hybrid hardware-software detection with hardware recovery.

The Resilient-System Design Team (RSDT) attempts to manage faults entirely in hardware by adding mechanisms for testing, monitoring, and adaptive recovery [4]. While effective for general-purpose computing systems, this approach is overly restrictive for emerging applications with few side-effects and ignores application error tolerance.

Finally, the Liberty Research Group proposes transparent software-based detection and recovery through compiler instrumentation [8, 33]. This software-only approach can be readily deployed in commodity hardware but has high performance overheads.

Application Error Tolerance. A variety of studies have attempted to quantify application tolerance to errors [6, 11, 22, 23, 42]. In contrast to Relax, they allow errors to affect program state rather than discard them. However, the general findings are that control flow and memory operations, which together constitute a large percentage of these applications, remain intolerant to errors. As a result, these studies ultimately advocate for various forms of detection and/or recovery. The only technique that incorporates neither detection or recovery involves manually identifying "soft" computations and allowing only the backwards slice of these computations to fail [23]. These instructions can in some cases account for more than half of an application's dynamic instruction stream, but in general the technique by itself does not scale well beyond fault rates of more than $1e^{-6}$, and even this technique would still require changes to the ISA and compiler for the software to communicate information on what is a soft computation to the hardware. The evident conclusion is that arbitrary and uncontrolled failure is not generally feasible.

10. CONCLUSION

As CMOS technology scales, hardware reliability is becoming a primary design constraint. While languages, ISAs, and microarchitectures continue to maintain the illusion of the transistor as a perfect switch, VLSI circuits, CAD, and manufacturing layers of the silicon stack are under tremendous pressure to maintain this illusion. Emerging applications provide an opportunity to mitigate these CMOS scaling constraints by relaxing the burden of fault recovery on hardware.

This paper presented the Relax framework, which relaxes architectural semantics to help simplify CMOS scaling by removing the illusion of perfect hardware. Specifically, we proposed a handful of simple extensions to the programming language, compiler, ISA, and microarchitecture levels that simplify hardware design by enabling efficient software-level recovery of hardware faults. We constructed a spectrum of language models combining retry and discard behaviors with coarse and fine recovery granularities to enable flexible application handling of errors.

We showed that PARSEC applications are easily relaxed for more than 70% of their execution with only a handful of source-line modifications required, and that significant further opportunity exists. Applying the framework to allow timing errors due to process variations, we show that, applications are up to 20% more energy-efficient. Most importantly, the correctness requirements of hardware are reduced. Overall, the Relax framework enables flexible and efficient handling of hardware reliability through multiple levels of the system stack, instead of placing all the burden on hardware alone.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the Vertical group for comments and the Wisconsin Condor project and UW CSL for their assistance. Many thanks to Mark Hill and Guri Sohi for several discussions that helped refine this work. Support for this research was provided by NSF CAREER award #0845751 and Toshiba corporation.

12. REFERENCES

[1] Google performance tools. http://code.google.com/p/google-perftools/.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *MICRO '03*, pages 205–216.

[3] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO '03*, pages 423–434.

[4] T. Austin, V. Bertacco, S. Mahlke, and Y. Cao. Reliable systems on unreliable fabrics. *IEEE Design & Test of Computers*, 25(4):322–332, 2008.

[5] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In R. Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[6] J. Bau, R. Hankins, Q. Jacobson, S. Mitra, B. Saha, and A. Adl-Tabatabai. Error resilient system architecture (ERSA) for probabilistic applications. In *SELSE '07*.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*, pages 72–81.

[8] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *DSN '06*, pages 83–92.

[9] M. de Kruijf, S. Nomura, and K. Sankaralingam. A unified model for timing speculation: Evaluating the impact of technology scaling, CMOS design style, and fault recovery mechanism. In *DSN '10*.

[10] M. de Kruijf, S. Nomura, and K. Sankaralingam. Design, modeling, and evaluation of the Relax

architectural framework. Technical Report TR-1672, University of Wisconsin-Madison, Department of Computer Sciences, 2010.

[11] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE '09*.

[12] 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 '03*, pages 7–18.

[13] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 176–187, 2008.

[14] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *PACT '07*, pages 213–224.

[15] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing processors for timing speculation from the ground up. In *HPCA '09*, pages 213–224.

[16] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09*, pages 140–151.

[17] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA '08*, pages 213–224.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07*, pages 211–222.

[19] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07*, pages 162–173.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88.

[21] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS '08*, pages 265–276.

[22] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA '07*, pages 181–192.

[23] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *Workshop on Architectural Support for Gigascale Integration*, 2006.

[24] 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 '05*, pages 190–200.

[25] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[26] F. Mesa-Martinez and J. Renau. Effective optimistic-checker tandem core design through architectural pruning. In *MICRO '07*, pages 236–248.

[27] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA '02*, pages 99–110.

[28] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA '03*, pages 129–140.

[29] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *ISWC '06*, pages 182–188.

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

[31] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ISCA '09*, pages 93–104.

[32] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA '02*, pages 111–122.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. on Architecture and Code Optimization*, 2(4):366–396, 2005.

[34] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN '08*, pages 70–79, 2008.

[35] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. VARIUS: A model of process variation and resulting timing errors for microarchitects. *IEEE Trans. on Semiconductor Manufacturing*, 21(1):3–13, 2008.

[36] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO '06*, pages 223–234.

[37] D. J. Sorin. *Fault Tolerant Computer Architecture*. Morgan & Claypool, 2009.

[38] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02*, pages 123–134.

[39] V. Sridharan, D. A. Liberty, and D. R. Kaeli. A taxonomy to enable error recovery and correction in software. In *Workshop on Quality-Aware Design*, 2008.

[40] 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.

[41] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS '95*, page 22.

[42] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE '06*.