Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance

Michael D. Powell*, Arijit Biswas*, Shantanu Gupta†1, and Shubhendu S. Mukherjee*

*SPEARS Group
Intel Massachusetts
Hudson, MA 01749
{michael.d.powell,arijit.biswas,shubu.mukherjee}@intel.com

†Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48105
shangupt@umich.edu

ABSTRACT

The incidence of hard errors in CPUs is a challenge for future multicore designs due to increasing total core area. Even if the location and nature of hard errors are known a priori, either at manufacture-time or in the field, cores with such errors must be disabled in the absence of hard-error tolerance. While caches, with their regular and repetitive structures, are easily covered against hard errors by providing spare arrays or spare lines, structures within a core are neither as regular nor as repetitive. Previous work has proposed microarchitectural core salvaging *to exploit structural redundancy within a core and maintain functionality in the presence of hard errors. Unfortunately microarchitectural salvaging introduces complexity and may provide only limited coverage of core area against hard errors due to a lack of natural redundancy in the core.*

This paper makes a case for architectural *core salvaging. We observe that even if some individual* cores *cannot execute certain operations, a CPU* die *can be instruction-set-architecture (ISA) compliant, that is execute all of the instructions required by its ISA, by exploiting natural cross-core redundancy. We propose using hardware to migrate offending threads to another core that can execute the operation. Architectural core salvaging can cover a large core area against faults, and be implemented by leveraging known techniques that minimize changes to the microarchitecture. We show it is possible to optimize architectural core salvaging such that the performance on a faulty die approaches that of a fault-free die--assuring significantly better performance than core disabling for many workloads and no worse performance than core disabling for the remainder.*

Categories and Subject Descriptors

B.8.1 Reliability, Testing, and Fault Tolerance, C.1.0 Processor Architectures

General Terms

Reliability, Performance

Keywords

Reliability, Hard Errors, Redundancy, Core Salvaging

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

1 INTRODUCTION

Current and future multi-core CPUs achieve high core counts by increasing the size of the processor die. As die size and transistor density grow, the susceptibility of these processors to hard faults grows as well. Hard faults result in a permanent defect, such as a bit stuck at a single logical value, and threaten yield, performance, and reliability of these multi-core CPUs. Hard faults, can occur in manufacturing or manifest later as wear-out faults in the field. Hard faults that result from manufacturing are generally uniformly distributed throughout the die (barring any especially susceptible structures), while hard faults caused by wear-out generally manifest on devices along wear-out susceptible paths. Defects detected at manufacturing time result in lost sales from either reduced yield (throwing the die away) or reduced capacity (selling the die with smaller caches or fewer cores). Defects detected in the field using future fault detection and isolation technologies [22] could necessitate substantial performance degradation to maintain reliable operation by disabling cores or shrinking caches.

Multi-core CPUs devote a large fraction of die area to regular memory structures, particularly caches. Fortunately, caches can be protected from manufacture-time defects using well-known techniques such as array sparing [27], line sparing, and error-correcting codes (ECC) [5]. These protection techniques *cover* the caches' area, meaning that the CPU is not vulnerable to reduced sales price or large performance loss in the event of (a reasonable number of) defects in that die area. However, effective coverage of caches leaves the remainder of the die as the major source of defect vulnerability. The bulk of this remainder is CPU cores.

Covering cores against defects is a challenge. One obvious solution, which we define as *core disabling,* turns off defective cores, resulting in the previously-mentioned problems of reduced sales price and/or reduced performance, depending on when the defect is discovered. Alternatively *core sparing*, providing extra cores that are kept in standby in case of defects, consumes precious die area while providing no performance or economic benefit in a non-defective die.

Current and near-term high-performance CPUs have small enough core counts, in the range of four to eight [13], that losing even one core can degrade performance substantially. Even potential future designs with dozens of cores [11] do not obviate the need to address defect vulnerability because these designs could experience defects in multiple cores, triggering large losses. In addition, heterogeneous designs like IBM's Cell [9], could be vulnerable if one of the few large, complex cores is unavailable due to defects.

1. This work was performed while Shantanu Gupta was an intern in the FACT Group within SPEARS at Intel Massachusetts.

Table 1: Redundancy and core salvaging terminology.

Term	Definition
Redundancy	
Redundancy	CPU can function correctly without a resource
Natural redundancy	CPU can function correctly without a resource because of pre-existing alternate (spare) resources
Artificial redundancy	CPU can function correctly without a resource because a spare resource has been added
Microarchitectural redundancy	intra-core redundancy: core can function without a resource and/or a spare exists in the core
Architectural redundancy	inter-core redundancy; core can function without a resource by using a spare on another core
Hybrid hardware redundancy	hybrid of artificial microarchitectural redundancy and architectural redundancy
Salvaging	
Core disabling	turning off a defective core and utilizing a CPU with fewer functional cores
Core sparing	alternate cores on standby, used only to replace defective cores; functional-core count is constant
Microarchitectural core salvaging	exploiting microarchitectural redundancy to make a defective core functional
Architectural core salvaging	exploiting architectural redundancy to make a defective core functional
Hybrid core salvaging	exploiting hybrid hardware redundancy to make a defective core functional
Replication of functionality	
Non-replicatable function	A function that can execute on only one resource (resource is not redundant)
Replicatable function	A function that can execute on more than one resource (resource *may* be redundant)

A more desirable alternative to core disabling and core sparing is *core salvaging*, which allows defective cores to continue to operate. Proposed techniques for *microarchitectural core salvaging* disable defective execution pipelines [19] or schedule operations on alternate or spare resources [23, 21] to avoid utilizing the defective area. These techniques require changes to the microarchitecture and introduce complexity. In addition, techniques that rely on disabling resources can recover only the *naturally redundant* fraction of core area that already exists due to performance-driven replication within the core. These are typically limited to structures that can be used in parallel, such as integer arithmetic-logic units (ALUs). The larger fraction of non-redundant execution resources--such as multipliers, dividers, and even the result busses between units--cannot be covered because functionality would be lost. Creating *artificial redundancy* by adding spare resources to cover non-replicated functions faces the same problem as core sparing--increased complexity and die area with no benefit in defect-free cores. Table 1 provides definitions of the terminology used throughout the paper.

We observe that even if some individual *cores* cannot execute specific instructions, a CPU *die* can be instruction-set-architecture (ISA) compliant, that is execute all of the instructions required by its ISA, by exploiting existing natural *cross-core redundancy.* Accordingly, we suggest a new *architectural (core) salvaging* mechanism that 1) covers a significant fraction of the core area and 2) requires only small changes to the microarchitecture. Nearly the entire execution hardware of the core, including non-replicated resources, can be covered against defects. Assuming that defects are known a-priori, as will be discussed in Section 2, only a few changes to the core are needed to detect un-executable operations and support thread migration. Instead of addressing a defective core's inability to execute an instruction using the microarchitecture, we propose using hardware to migrate the offending thread to another core that can execute the instruction. If the other core was busy, the migration becomes a thread swap. By implementing the thread migration in hardware, this mechanism can be made transparent to the operating system (O/S) and user.

Architectural core salvaging avoids the main problem of core disabling by allowing faulty cores to contribute to overall throughput. How much faulty cores contribute to throughput depends on the frequency of non-executable instructions with core-disabling providing a lower bound (at zero throughput for defective cores). We divide the possibilities into three cases. First, in the case of infrequent instructions--those that may occur only a few times in a workload or in a few clusters of occurrences, such as floating point divide or square root--performance loss compared to a fault-free die can be negligible. Even if all threads executing on the die use instructions that are non-executable by one or a few cores, migration overhead can be amortized over the large gaps (e.g., tens of thousands of cycles) between those infrequent instructions since there are only a few migrations. Second, in the case of frequently-occurring instructions that are used by only some executing threads (e.g., heterogeneous multi-programmed or multi-threaded workloads), we utilize migration policies to find quickly a stable thread schedule where the defective core executes a thread that does not utilize un-executable instructions. Once the schedule stabilizes, there will be no performance loss compared to a fault-free die. Third and finally, for instructions that a workload uses frequently in all threads, we fall back on core disabling to avoid thrashing and bound our performance to be no worse than that of core disabling. We also avoid pathological performance loss by not utilizing cores incapable of executing certain critical instructions (e.g., load, store, branch, integer arithmetic).

Architectural core salvaging offers advantages over microarchitectural core salvaging by being simpler, requiring fewer changes to the core, and covering a larger area. Though the idea of architectural salvaging is in itself simple, potential pitfalls of cross-core migrations and execution on defective cores must be carefully avoided to maintain high performance and user transparency.

The main contribution of this paper is to make a case for *architectural core salvaging* to exploit natural *architectural* redundancy, which we define as cross-core redundancy, and cover a large core area against faults without substantial changes to the microarchitecture. Specifically:

- We describe how leveraging existing CPU capability to enable thread migration can minimize architectural core salvaging's intrusion into the core, simplifying the implementation.
- We show it is possible to optimize architectural core salvaging to have performance on a faulty die near that of a fault-free die--significantly better than core disabling for many workloads and no worse for the remainder.
- We describe architectural and hybrid core salvaging opportunities to cover approximately 30% of the vulnerable core area compared to 10% for microarchitectural salvaging.

The rest of this paper is organized as follows. In Section 2 we discuss related work on defect detection and exploiting redundancy. Section 3 explains some of the limitations of microarchitectural redundancy. In Section 4 and Section 5 we introduce our architectural core salvaging technique. Section 6 explains our experimental methodology, and Section 7 presents our results. We conclude with Section 8.

2 RELATED WORK

In this section, we briefly discuss techniques on detecting and isolating defects caused by hard errors. Next, we discuss previous defect tolerance work. Fault tolerance techniques generally consist of two distinct and necessary pieces, fault detection/isolation and error correction/recovery. This paper focuses on the latter aspect of correction/recovery via techniques to exploit redundancy regardless of the underlying defect-detection technique. While detecting and isolating faults, either at manufacturing time or in the field, are necessary components of any fault-tolerance scheme and may add additional complexity, they are beyond the scope of this work.

2.1 Defect Detection and Isolation

Techniques for detecting defects can be divided into manufacturing-time and run-time methods. We first discuss manufacturing time detection. Manufacturing test primarily focuses on identifying defective parts, but those detection techniques can be expanded to isolate faults to determine if a particular defect might be covered by redundancy.

Manufacturing test has an advantage over run-time detection because the tester has access to scan chains and other debug mechanisms which provide fine-grain visibility into micro-architectural CPU state. Manufacturing test to detect (but not isolate) defects is a well understood topic [4]. Some previous proposals aim to aid manufacturing test in isolating defects. Rescue, proposed in [19], proposes altering the microarchitecture to increase visibility of scan. They propose increasing intra-cycle-independence of microarchitectural structures throughout a pipeline so defects can be isolated to the logic between two successive pipeline latches.

Run-time isolation is more challenging than manufacturing-time isolation. One proposal for run-time isolation is BlackJack [20], which exploits simultaneously-redundant threads on an SMT, previously used to detect soft errors, to detect defects. Bower et al. in [3] propose using DIVA-checkers, small auxiliary cores that check committed instructions [1], for defect isolation. Constantinides et al. in [6] propose a virtualization layer between the operating system and the hardware to introduce periodic special instructions for defect isolation.

2.2 Defect Tolerance

2.2.1 Using Translation or Virtualization

There have been proposals to tolerate defects via binary translation or virtualization. Detouring [16] is an all-software approach for defect tolerance in simple cores that translates code to avoid defects in execution units, register files, and instruction and data memory. While well-suited to simple cores, high-performance x86 cores typically avoid binary translation layers. Joseph [15] proposes using a virtualization layer to tolerate coarse-grained execution-cluster defects in an Alpha 21264 microarchitecture via either instruction emulation or thread migration.

2.2.2 Using Microarchitectural Salvaging

There have been several proposals to exploit microarchitectural redundancy, or add extra redundancy, to tolerate defects via microarchitectural salvaging. Rescue [19] combines their fault-isolation mechanism with de-mapping failed redundant resources. Rescue achieves high coverage but requires alterations to many components to create exploitable redundancy, such as segmenting the issue logic into two halves and forbidding same-cycle communication between the halves. These alterations add complexity and may add performance degradation.

Bower et al. [2] utilize microarchitectural salvaging, which they call self repair, in array structures. Srinivasan et al. [23] analyze the performance impact of both moving execution to duplicated resources and gracefully degrading performance by disabling failing redundant resources.

Core Cannibalization [18] allows one core to borrow resources from another core at the pipeline-stage level. While cannibalization exploits a form of cross-core redundancy, we classify it with intra-core microarchitectural salvaging because the additional interconnect required to support borrowing closely couples the cores. Such interconnect also presents performance and complexity challenges.

Shivakumar et al. in [21] identify microarchitectural redundancy in an Alpha 21264. They exploit redundancy by de-mapping failed combinational units and de-mapping failed entries in small arrays such as the reorder buffer and register files. They report high coverage, but as we discuss in the next section, their coverage estimates may be too optimistic.

3 LIMITATIONS OF MICROARCHITECTURAL REDUNDANCY

In this section, we discuss how defect coverage derived from microarchitectural redundancy may be limited in modern CPUs. Recall from Section 1 that because of existing coverage techniques for large memory structures, the main defect vulnerability lies in the cores, and that successful defect-tolerance techniques must cover a substantial fraction of core area. We first discuss coverage

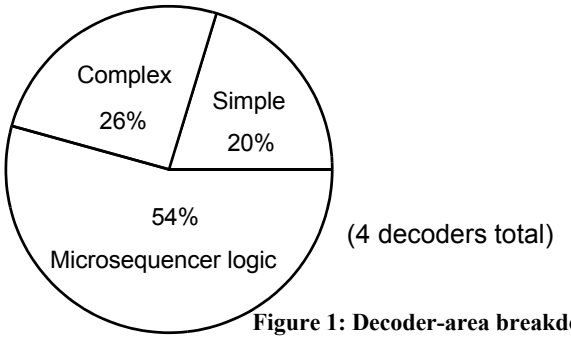

Figure 1: Decoder-area breakdown.

Non-replicated instruction classes (A)		
branch	load	store
control reg.	slow ALU	int mult.
int divide	simd mult.	simd other
fp load	fp store	fp mult.
fp divide	fp ROM	fp shuffle
fp other	fp add	fp other

Replicated instruction classes (B)	
logical	simd_shift
vector sse	condition code
int shuffle	simd shuffle
Replicated instruction classes using solely redundant structures (C)	
int ALU	shift
vector inst.	extended simd shuffle

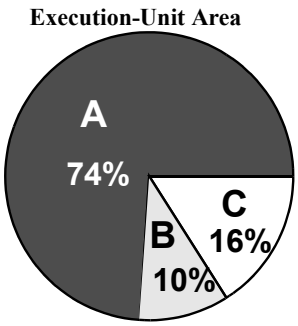

Figure 2: Example of execution-unit redundancy in an x86 core.

challenges in exploiting microarchitectural redundancy in combinational logic, and then discuss challenges for sequential logic contained in small arrays. Finally, we discuss general concerns for microarchitectural redundancy which impact all structures.

We do not discuss large arrays, including caches, translation lookaside buffers (TLBs), and register files, because these arrays can already be covered against defects by introducing artificial redundancy in the form of spare rows, spare columns, spare arrays [27], and error-correcting codes as mentioned in Section 1.

3.1 Limited coverage of combinational logic

While certain combinational structures are typically redundant in superscalar CPUs, many are not. Some architectures may also have more redundancy than others. We discuss the impact of certain microarchitectural techniques in two portions of the CPU pipeline, the front end and back end. The front end includes the fetch and decode logic while the back end includes the rename tables, reservation stations, execution resources and re-order buffer.

3.1.1 Front End

Rescue [19] considers each of the instruction decoders in a pipeline to be equivalent and thus redundant. However, an instruction-set-architecture (ISA) such as x86, which includes both simple instructions and complex instructions which decode into microcode flows, typically utilizes a single complex decoder and several simple decoders. Only the simple decoders are redundant. Unfortunately, the single complex decoder can be larger in area than all of the simple decoders combined. If the microsequencer logic (excluding microcode ROM arrays, which could be protected by ECC) is included as part of the decoder, the simple decoders represent an even smaller fraction. The relative area of all simple decoders, the complex decoder, and the microsequencer logic of an Intel® Core-2™-like processor are shown in Figure 1.

3.1.2 Back End

Multiple equivalent execution units in a superscalar CPU are the most typically cited example of microarchitectural redundancy. It is easy, however, to overestimate execution-unit redundancy. Execution-unit redundancy is extremely dependent on both ISA and microarchitectural symmetry. (i.e., are there many general units which can execute most instructions?) When considering execution-unit redundancy, it is important to determine the redundancy of *structures* in addition to *replicatability* of *instructions,* which is simply whether an instruction can execute on more than one structure, even if those structures are not redundant. Only structural redundancy increases defect coverage, regardless of instruction replication.

Some structures, such as input muxes and result busses, may be utilized across multiple classes of instructions. However, even if some of the instructions are replicated instructions that can execute on multiple structures, the structures themselves may *not* be redundant because these structures are also required for some other class of instruction. For example, there may be multiple integer adders but only one that is capable of generating addresses for loads. As a result, neither the adder that generates load addresses, nor any of the input/output circuitry for that adder, are redundant even though there may be other similar adders and other input/output circuitry. While it may seem desirable for reliability to create a more general architecture with more inherently redundant structures, doing so can lead to complexity that can impact performance and/or power, as discussed in Section 1.

For execution-unit redundancy, Shivakumar [21] takes advantage of the fairly symmetric, clustered 6-instruction-issue Alpha 21264 microarchitecture which contains duplicate integer and floating-point clusters. However, such wide-issue cores have fallen out of favor in part due to complexity and power concerns.

To illustrate another possibility, we analyzed the execution-unit redundancy in an Intel® Core-2™-like x86 microarchitecture, as shown in Figure 2. Note that execution-unit redundancy is highly implementation specific, and this analysis is just one representative example. The instruction (micro-op) classes are separated into two bins shown in the table, the replicated classes which can be executed on more than one functional unit (B and C), and non-replicated classes which cannot (A). To complete the redundancy analysis, however, we need to map each class of instruction to the structures it uses to determine which structures are redundant.

To complete the mapping, we use activity information from a detailed Architectural Level Power Simulator (ALPS) like architectural power model that tracks activity and power for over 300 functional-unit-blocks per core, as described in [10, 17]. This model maps occurrences of each class of instruction to microarchitectural structures. Structures that are used for only replicated instructions are themselves considered redundant. The pie chart shows the relative areas of redundant execution structures (executing replicated instructions in C) and non-redundant execution structures (execution non-replicated instructions in A and replicated instructions in B) within the total execution-unit area. As shown in the figure, the redundant structure area is small compared to the total area, and the number of replicated instruction classes that execute on redundant structures is much smaller than the number of replicated instructions.

3.2 Limited area coverage in small arrays

Many of the structures in both the front and back ends of modern CPU cores are small non-cache arrays that are generally not covered by cache redundancy techniques (e.g., spare rows, spare columns, etc.) because of their small size and the high relative overhead of such coverage. Small arrays, particularly those that

use circular queues, can be easily covered using microarchitectural salvaging by enhancing their decoder and pointer logic to skip defective entries. Shivakumar [21] showed that structures such as the ROB can lose a few entries with minimal loss of performance. Our own studies on an Intel® Core-2™-like processor (detailed later in Section 6) concur that losing several entries in either the instruction decode queue, reservation stations, or re-order buffer has minimal performance impact (on average less than 1%, and at worst less than 10%).

Small-array area coverage estimates can be misleading. Tolerating defective array entries covers only the static RAM (SRAM) cells and wordlines, but not the supporting logic of bitlines, sense amps, decoder logic, and array-pointer logic. While in caches the area of memory cells may dwarf that of the support logic, that is not the case in small arrays where the support logic is a substantial fraction of the area.

Figure 3 shows area breakdowns for SRAM cells and support logic for several structures in an Intel® Core-2™-like microarchitecture. To estimate area, we use CACTI version 5.3 [26] configured for a 45nm technology and aggressive interconnects. We consider the *area efficiency* of the arrays, as reported by CACTI, as a measure of redundant area. Area efficiency is the fraction of structure area devoted to memory cells, and excludes non-redundant components like decoders and sense amplifiers. As representative small arrays, we model a decoded-instruction queue and reorder buffer. We also show the data array of a typical L1 cache for a reference on potential redundancy in large data arrays. The redundant area in the small arrays are both less than 20%, while the redundant area in the cache data array is 60%. This result illustrates the limited area coverage of microarchitectural redundancy in small arrays. Array salvaging techniques cover a large portion of a cache array but are less effective for small arrays. It is also worth noting that the example structures are all RAMs with no additional support logic other than that needed to read and write data. RAMs with additional support logic and content-addressable memories (CAMs) would show substantially less redundancy.

3.3 General concerns

Another key concern with microarchitectural redundancy is that nearly every redundant structure requires a slightly different salvaging mechanism. This complexity was illustrated in Rescue [19], which described their techniques pipe stage by pipe stage, and by Shivakumar et al., which exploits three meta-categories of redundancy (component-level, array, and dynamic-queue) across 20 structures. Covering these structures might require structure-specific implementation details. For example, the dynamic-queue redundancy in the integer instruction window, a CAM, is unlikely to be the same as that in the register file, a RAM.

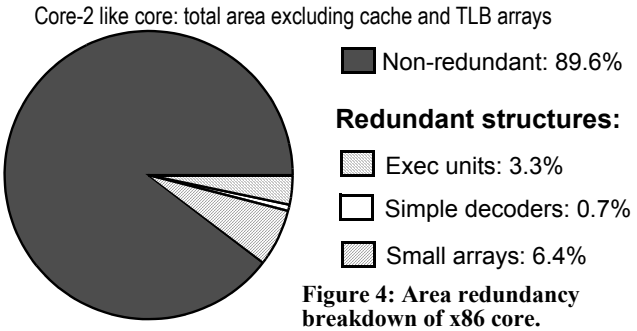

Figure 4: Area redundancy breakdown of x86 core.

Unfortunately, there seems to be no one-stop solution to exploit microarchitectural redundancy in a large area of the core. Worse, when taking into consideration the true structure redundancy as described above in Section 3.1 and Section 3.2, the total fraction of area covered by microarchitectural redundancy is not high. Figure 4 breaks down the area redundancy of our Intel® Core-2™-like CPU core based on the discussion of the previous subsections. The total area of the core excludes cache arrays and TLB arrays, which we assume can use existing salvaging techniques. The figure shows the non-redundant area and the redundant structure area for each of the execution units, simple decoders, and small arrays. The small-array area includes the reservation stations, fetch queue, load buffer, and store buffer area in addition to the reorder buffer and decode queue areas discussed in Section 3.2. Combined, only about 10% of the non-cache core area is covered. This data indicates that in order to achieve high coverage, we need to look beyond mere microarchitectural redundancy.

4 CORE SALVAGING WITH ARCHITECTURAL REDUNDANCY

Architectural redundancy provides an opportunity for increased coverage by relaxing the requirement that each individual core be fully functional and using the other cores to complete the missing functionality. This relaxation greatly increases the opportunity for redundancy coverage, because non-replicated structures within a core can be considered non-essential and be covered against defects by migrating threads that use these defective structures away from a defective core.

In this section, we present the potential of architectural redundancy. We then discuss the use of architectural redundancy through architectural core salvaging to increase hard-error coverage, particularly in combinational logic. Architectural redundancy is well

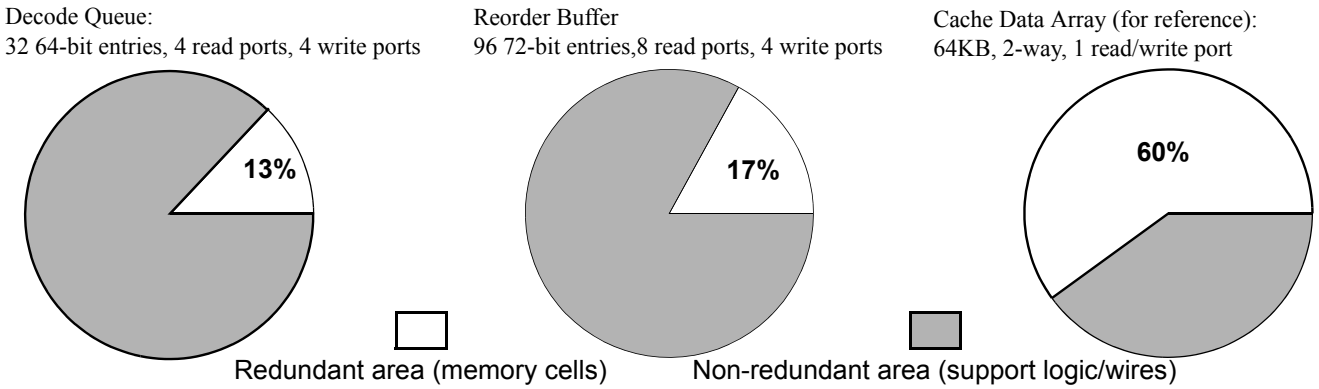

Figure 3: Redundant and non-redundant area of small-array structures and a cache data array.

Table 2: Core salvaging potential.

Cores per die	Die throughput relative to defect-free die if:		
	1 core deactivated	1 core loses x% throughput	
		x=10%	x=25%
1	0.00	0.90	0.75
2	0.50	**0.95**	0.88
4	0.75	**0.98**	0.94
6	0.83	**0.98**	**0.96**
8	0.88	**0.99**	**0.97**
12	0.92	**0.99**	**0.98**
16	0.94	**0.99**	**0.98**
20	0.95	0.99	0.99

suited to combinational logic, which may be used for only a portion of instructions but less suited to covering small arrays in the pipeline (such as the decode queue discussed in Section 3.2) because these structures are critical for pipeline functionality. In Section 5 we will introduce a hybrid of architectural and microarchitectural redundancy that can cover critical array structures. This section also discusses implementation options and optimizations necessary for architectural redundancy to maintain high performance in the presence of defective cores.

4.1 Potential of Architecture Salvaging

Architectural salvaging makes sense in a multicore design where the number of cores is small enough to warrant trying to salvage (rather than deactivate) a defective core but large enough to amortize the performance degradation due to migrating threads away from a defective core. In this subsection, we explore that design space.

Table 2 depicts a simple analytical model of overall throughput on a multicore die for core counts ranging from 1 to 20. The columns show throughput relative to a defect-free die for various defect-tolerance mechanisms. The second column shows relative throughput if 1 core is deactivated (so, for example, the throughput is 0 for a 1 core die). The third and fourth columns show relative throughput if 1 *core's* throughput is degraded 10% and 25% due to defects, such as those tolerated by architectural salvaging.

Assuming we are willing to tolerate a 5% degradation in overall die throughput, the shaded cells represent the core counts for which architectural salvaging is an effective solution. Once a die has 20 cores, core deactivation reduces throughput only 5%. At the other extreme of small core counts, throughput degradation from architectural salvaging can be intolerably high, such as in the cases with less than 6 cores in the fourth column. Given the depicted potential throughput degradations for defective cores, the sweet spot for architectural salvaging is die with more than 4 but fewer than 20 cores, which is similar to proposed designs for the next few generations of microprocessors [13].

It is worth noting that in cases of multiple defective cores, architectural salvaging may be worthwhile at higher core counts. Multiple defects may be a concern in future unreliable processes, but in this paper, we limit our modeling to a single defective core.

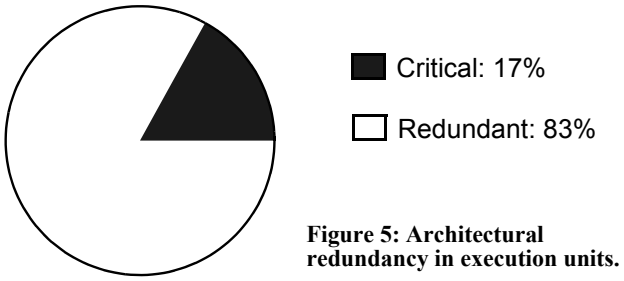

Figure 5: Architectural redundancy in execution units.

4.2 Redundancy in Combinational Logic

The key limitation in covering combinational logic, as discussed in Section 3.1, is that so much logic is not replicated. Architectural redundancy increases opportunities for coverage in both the back-end execution units and front-end instruction decoders.

4.2.1 Execution Units

For architectural redundancy, the coverage criteria is no longer which structures are redundant (as discussed in Section 3.1) but which structures are not critical for basic functionality. As discussed in Section 1, we define a minimal set of instructions for a core to be considered functional. These include loads, stores, branches, and arithmetic integer ALU operations. (While somewhat arbitrary, this list is based on intuition and high-occurrence of certain instructions.) Only the structures necessary to execute those operations are considered critical; everything else may be considered redundant. Figure 5 breaks down execution-unit area in our Intel® Core-2™-like CPU into that for critical instructions and that which is redundant. Coverage opportunities for architectural redundancy are substantially higher (83%) than the microarchitectural redundancy opportunities shown earlier in Figure 2 (16%).

However, as we explore later in Section 7, we may not want to exploit all of the available redundancy. Some operations beyond the critical set may be used so frequently as to justify disabling any core that is incapable of executing them.

4.2.2 Instruction Decoders

The instruction decoder also contains a large amount of non-replicated area. Recall from Figure 1 that the complex-instruction decoder and microsequencer logic are substantially larger than the simple instruction decoders and are not replicated, and only the simple instruction decoders are microarchitecturally redundant. Because certain complex instructions are rare in many applications, architectural redundancy might reverse the situation by making the complex decoder and microsequencer logic partially redundant as well.

4.3 Implementation

This section discusses how we implement architectural core salvaging by making minimal additions to the core and by leveraging existing CPU capabilities.

4.3.1 Minimal Core Changes

One goal of architectural salvaging is to minimize changes to the core. The two main requirements for core salvaging using architectural redundancy are 1) detecting the presence of instructions that cannot execute, and 2) transferring the architectural state of the thread to and from the core.

Detecting defective instructions can be done with a comparison at instruction decode. The system can use a lookup table, either fused at manufacturing or programmed in the field, to identify un-

executable instructions. Since the thread migration must occur before any defective instructions commit, detecting at decode allows adequate time to trigger a stall and initiate a migration without placing the lookup on any critical paths.

Transferring the architectural state requires transferring the architectural registers to and from a buffer off of the core. Fortunately, this capability already exists as part of deep-sleep power states such as *core C6*, which moves state off of a core to an on-die SRAM for power savings [12]. The core C6 array used to store an x86 core's *micro*architectural state is about 10KB [8]. Leveraging this capability for architectural core salvaging merely requires adding support (outside the core itself) for migrating the state from one core to another.

4.3.2 Migration and Overhead

Assuming all cores are busy, each thread migration is actually a thread swap between a salvaged core and another core. State migration between cores can occur over whatever interconnect is already present (bus, etc.), or via a specialized interconnect. The bandwidth requirements to migrate the architectural state are a few kilobytes in the case of the low-register-count x86 architecture. (Although the core C6 array is 10KB, the array was designed to hold all of the microarchitectural state for a core [8]. We need to store and to migrate only the architectural state.) The overhead of this migration may be on the order of many tens to a few hundred cycles, but should be amortized as long as migrations are infrequent. The overhead of warming up caches and branch predictors after a migration will be similarly amortized.

4.3.3 Operating System Transparency

To keep our technique entirely hardware-based, we propose keeping salvaging-based migrations transparent to the operating system (O/S). Doing so requires "fooling" the O/S into believing that a migrated thread is operating on the same logical core, which we accomplish by leveraging the existing Advanced Programmable Interrupt Controller (APIC) ID number. The O/S already identifies cores through their APIC ID number. While conventionally this number is fixed, it is a small change to make it programmable, allowing the APIC ID to migrate with a thread.

It may be possible to optimize architectural salvaging using the O/S, but these optimizations, while interesting, are beyond the scope of this work.

4.4 Optimizations

In this section, we discuss hardware optimizations to architectural salvaging to avoid performance loss.

4.4.1 Migration Policy

The core-selection policy for a thread that must be migrated from a defective core must be chosen carefully. Assuming all cores are busy, each migration is a thread swap between two cores. The policy should ensure that in the event of repeated migrations, a small subset of threads do not bounce on and off the defective core. Allowing the defective core to swap threads with all cores ensure that if a thread exists that does not utilize the defective components, the CPU will eventually settle on a stable thread assignment.

A simple policy that meets this goal is round robin. Under this policy, if core N is defective, it will swap with each other core before returning to swap with a core again. A more complex policy might use performance-monitoring registers to maintain a thread instruction profile for each core and intelligently select threads that are least likely to trigger such migrations. Complex policies such as this one may be helpful at reducing migrations in the event that more than one core is defective, because they can avoid migrating a thread onto a core that will likely migrate again soon.

4.4.2 Triggering Migrations

Thread migration can be triggered as soon as the un-executable instruction is seen, at instruction decode, or just before the instruction would commit. The advantage of triggering migration at decode is that the process starts sooner. The advantage of triggering migration at commit is that decoding an un-executable wrong-path instruction will not force a needless migration. However, an effective branch predictor may obviate the need for triggering migration at commit by largely avoiding fetch and decode of wrong-path instructions.

4.4.3 Fall-back to Core Disabling

Some workloads may experience many frequent migrations because all threads utilize a defective structure often. In such cases, it may be better to fall back to core disabling and simply deactivate the defective core. If necessary, this fall-back can be made O/S transparent by maintaining the architectural state of the extra thread on-die and, at intervals near the O/S quanta intervals, rotating which thread is assigned to the non-executing core.

Fallback to core disabling can be triggered by maintaining a migration count over a certain time window. If, for example, a core is forced to migrate a thread more than 10 times per 100,000 cycles, the defective core can be temporarily disabled. Fallback could also be triggered if throughput on a core drops below some pre-determined level. While falling back to core disabling may degrade performance, particularly for parallel workloads, it is better than the alternative of the thread thrashing between cores.

Disabling migration implies stalling a thread that would be executing on the defective core, so we must carefully avoid potential deadlock conditions from starving that thread. These can be avoided by migrating and re-scheduling the stalled thread to a non-defective core on O/S-quanta-sized intervals, and stalling a thread that would otherwise execute. O/S quanta are long enough to amortize the overhead of the migration but frequent enough to ensure forward progress.

4.4.4 Running Less Than the Max Number of Threads

If fewer than the maximum number of threads are available to execute, the defective core should be left idle, and there is no throughput degradation due to the defect. While this case is pathological and quite simple, it may actually be quite common in multi-core systems. The policy can be implemented in an O/S-transparent fashion by simply assigning the lowest-priority APIC ID to the defective core. (Recall from Section 4.3 that the APIC ID is made programmable for architectural salvaging.)

5 HYBRID HARDWARE REDUNDANCY

Architectural redundancy alone can cover only structures that are not essential for basic pipeline functionality, leaving many structures uncovered. In this section, we describe a hybrid of architectural and microarchitectural redundancy which makes small changes to the core, like architectural redundancy, but covers defects in pipeline-critical structures, like microarchitectural redundancy.

5.1 Hybrid Redundancy Overview

Section 3.2 discussed limitations of microarchitectural redundancy in small arrays. Although microarchitectural-redundancy

opportunity is limited, architectural-redundancy opportunity seems even less because any of these small arrays are critical for pipeline functionality.

We propose the introduction of minimally functional, structure-independent, small artificially redundant secondary replacements for some of these primary small arrays. By minimally functional, we mean that a pipeline will continue to function correctly, although possibly with large performance degradation. By structure-independent, we mean that the secondary structure will have its own decoders and/or support logic, so that the entire primary structure is covered. The secondary structure could also achieve independence by being a stand-alone part of the primary structure. (e.g., a sub-array with its own decoders) Finally, we require that the secondary structure be small compared to the primary structure to avoid high area overhead and potential for additional defects. It is important to note that the proposed secondary structures are *not* full-scale copies of the primary structures.

If a primary structure is defective, we use the secondary structure to maintain core functionality. We use architectural salvaging to minimize the performance impact of the degraded core, relying on extensions of the optimizations discussed in Section 4.4. Instead of a simple round-robin migration policy, more intelligent thread-selection mechanisms that take into account the specific defects on a given core are likely necessary. We describe such policies next.

5.2 Examples of Hybrid Redundancy

In this subsection, we give examples of structures that can exploit hybrid redundancy and describe policies to mitigate performance loss associated with defects in these structures.

5.2.1 Branch Predictors

Branch prediction is not strictly necessary at all for functional correctness, although a minimally functional branch predictor is necessary to avoid pathological performance loss. To cover the primary branch predictor against defects, a simple bimodal predictor with only a relatively few entries can be introduced as a secondary structure. Such a predictor will still correctly predict the majority of branches; [14] showed prediction rates of over 90% for a 2KB bimodal predictor. To mitigate performance loss in a defective core using the secondary predictor, the thread selection policy may choose threads such that they either a) have extremely high branch prediction rates (and thus might be handled by even the simplest predictor) or b) have extremely low branch prediction rates (and thus will be handled poorly on any predictor). If threads are run for some time on both a defective and non-defective cores, the thread selector also might choose the thread with the smallest change in prediction-rate.

5.2.2 Load and Store Buffers

Memory re-ordering is not strictly necessary for functionality. If the load or store buffers are defective, the core can force all memory operations to execute in program order, covering the buffers and most of their support logic against defects. The thread selector can mitigate performance loss by either choosing a thread with few memory operations for the defective core, or similar to the branch-predictor scheme, choosing a thread that sees the least performance degradation in the absence of reordering.

5.2.3 Other structures

Other structures that can be covered using hybrid redundancy include the fetch queues, decode queues, and out-of-order execution buffers. The fetch and decode queues can be covered by adding minimally-sized secondary queues (i.e., sized at the fetch and decode width).

Covering the out-of-order buffers, specifically the reservation stations and/or reorder-buffer, is more ambitious. These structures are highly integrated into the pipeline, but it may be possible to function with defects if all instructions are forced to execute in program order.

6 METHODOLOGY

To analyze performance of using architectural core salvaging, we use a detailed execution-driven simulation of an Intel® Core-2™-like microprocessor core in the Asim [7] simulation environment. The parameters of our system, are shown in Table 3.

Our area estimates for memory arrays come from Cacti [26]. Our area estimates for cores and execution units are derived from block-level estimates from our target microprocessor core.

We assume a 100-cycle overhead to migrate architectural state between cores. This overhead is in addition to the overhead from draining instructions from the cores prior to migration.

We use applications from the SPEC CPU 2000 [24] and SPEC CPU 2006 [25] suites, in addition to multimedia and server workloads. Our server workloads consist of TPC-C, TPC-H, and specweb99. A summary of the number of benchmark runs in each group appears in the first two lines of Table 4. We run 8 copies of the same thread, providing a worst-case workload for our core-salvaging technique because all threads will need the same resources. (Realistic non-worst-case workloads would be heterogeneous, as discussed in Section 4.4.) Unless otherwise noted, we run each application for 5 million instructions after warming up cache state.

7 RESULTS

In this section we present our results. First we give an analysis of the potential for core salvaging in execution units with small performance degradation. Then we present our results for core salvaging in combinational structures. Due to space and computation limitations, we show a small subset of the potential redundancy available across many classes of instructions. Finally, we present examples of hybrid redundancy.

7.1 Opportunity

In this section, we present the opportunity for core salvaging in execution units to avoid performance degradation based on the

Table 3: System Parameters.

Instruction issue	4, out-of-order
I-cache	64KB 4-way
D-cache	64KB 8-way, 2 cycles, 2 load ports
Branch Predictor	Bimodal (512 entries) + Gshare (1024 entries)
Branch Target Buffer	4K entries; 16-way
Fetch / Decode queues	14/24 entries
Reservation Stations	32
Reorder Buffer Entries	96
Load/Store Buffers	50/24 entries
L2 cache	1MB, private, 8-way, 10-cycles
Cores	8
L3 cache	Shared, 8MB, 8-way

Table 4: Benchmarks with high and low fraction of 100K-instruction windows missing instruction class.

workload	spec int 2K		spec fp 2K		spec 2006		server		multimedia	
benchmarks	32		34		31		27		73	
fp div	gzip (1)	vpr (0)	sixtrack (1)	apsi (0)	milc (1)	gamess (0)	specweb99ssl -1 (1)	specjappser ver04 (0.42)	photoshop (1)	renderman ball (0)
fp mul	gzip (1)	vpr (0)	sixtrack (1)	lucas (0)	libquan-tum (1)	gromacs (0)	specweb99ssl -1 (1)	tpcc_yukon (0.21)	photoshop (1)	renderman ball (0)
fp rom	all are 1		wupwise (1)	facerec (1)	all are 1		db2_tpch (1)	tpcc_yukon (0.93)	photoshop (1)	videostu-dio (0)
i mul	bzip2 (1.0)	vpr (0)	swim (1)	facerec (0)	mcf (1)	astar (0)	specweb99ssl -2 (0.47)	tpcc_yukon (0)	renderman _ball-2 (1)	renderman _ball (0)
i div	bzip2 (1)	twolf (0)	galgel (1)	fma3d (0)	libquan-tum (1)	sjeng (0)	db2_tpch (0.84)	tpcc_sql_20 00 (0)	videostu-dio (1)	spec-apc-3dsmax (0)
i shuf	gap (1)	eon (0)	equake (1)	apsi (0)	per-lbench (1)	gamess (0)	all are 1		spec-apc-3dsmax (1)	cinema4dr 8_stairs (0)
si(md) shift	parser (1)	vpr (0)	art (1)	fma3d (0)	gcc (1)	gamess (0)	all are 1		spec-apc-3dsmax (1)	cinema4dr 8_stairs (0)
i slow	bzip2 (1)	vortex (0)	wupwise (1)	mesa (0.06)	bzip2 (1)	soplex (0.99)	db2_tpch (0.98)	tpcc_win2k (0)	rendermal _ball-2 (1)	photoshop _2 (0)

occurrence of instruction classes in the benchmarks. We expect a large number of benchmarks to use rarely or never certain classes of instructions, representing an opportunity to salvage cores that cannot execute those instruction classes.

For core salvaging, we are interested in extended execution periods that are absent certain instruction classes, providing an opportunity to execute on a defective core. We computed the mean, median, and standard deviation of distances between instructions of given classes (not shown), but found the numbers not particularly useful. Similar instructions tend to occur in bursts, making the statistics misleading. Instead, we computed the fraction of non-overlapping 100,000-instruction periods that lack specific instruction classes in a 10-million instruction (single-thread) run. 100,000 instructions represents a period long enough to execute a thread on a defective core while amortizing the overhead of a potential future migration.

Figure 6 presents statistics on instruction occurrence across our workloads. Across the x-axis are 8 instruction classes; the 5 workloads have a bar for each instruction class. The top graph shows the average fraction of 100,000-instruction intervals that are absent that instruction class for a particular workload. (A value of 1 means that instruction class never occurs while 0 means it occurs in every window.) The bottom graph shows standard deviation of the fractions.

Instruction occurrence varies across workloads, but many of the instructions are infrequent in several workloads. Floating-point divide is largely absent from all but spec2K-int and multimedia. Floating-point ROM instructions, which utilize the constant ROM, are rare except in multimedia.

An outlier among the shown instruction classes is integer multiplies, which occur frequently in every workload. Because integer multiplies occur in more than 50% of windows in all of the workloads except spec2K-fp, integer multiplies may be a candidate for addition to the critical instructions which a core must be able to execute to be considered functional, as described in Section 4.2.1.

Table 4 presents a deeper dive into the instruction occurrences, showing example benchmarks that are at the high occurrence rate and low occurrence rate for each workload. The majority of the low (and high) occurrences are at or near 0% (and 100%), indicating that while there are trends within each workload, as shown in Figure 6, most workloads still contain some benchmark with both frequent and infrequent occurrence of the instruction classes. Exceptions are that none of the server workloads utilize integer shuffle or simd shift, and the fp-rom is not used by either spec2k-int or spec 2006.

In the next section, we show the performance impact of having one core unable to execute these instruction classes as a result of core salvaging.

7.2 Core-Salvaging

In this section, we present throughput results for core salvaging when one core is unable to execute particular instructions. We expect larger area coverage than microarchitectural salvaging can

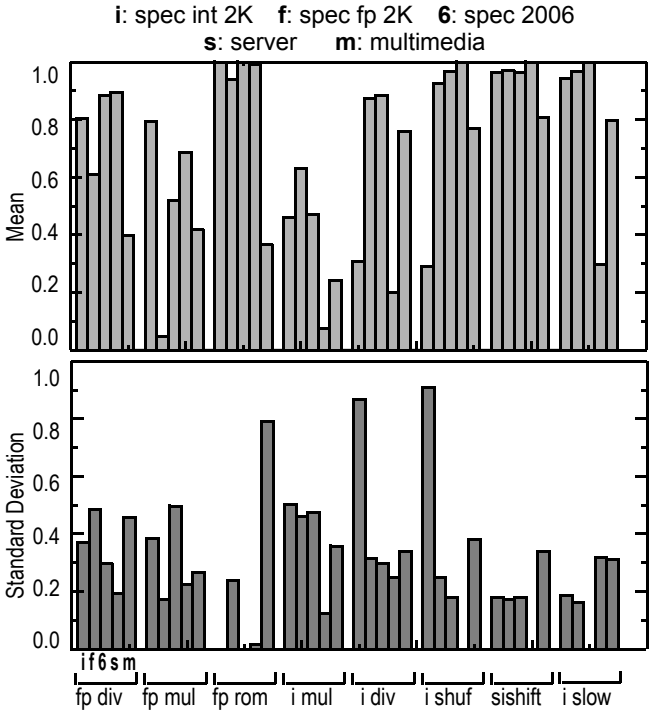

Figure 6: Fraction of non-overlapping 100K- instruction windows that do not contain an instruction class for 5 workloads.

provide. We expect throughput for core salvaging to be near that of a defect-free core for applications that use the un-executable instructions never or infrequently. For applications that use un-executable instructions frequently throughout their execution, we expect throughput similar to that of core disabling. We show behavior for homogeneous 8-thread workloads because they are more interesting and are worst-case. Heterogeneous workloads would quickly settle on a stable or near-stable schedule and would perform as well or better than homogeneous workloads.

Figure 7 (A) shows average throughput for each of our workloads relative to 8 defect-free cores (top bar) and 7 defect-free cores (bottom bar, equivalent to core disabling of one core). Each group of bars corresponds to disabling one or more instruction classes. From left-to-right, the first 7 are a subset of the classes in Section 7.1 The last three represent disabling both integer and simd shuffle (i,si-shuf), all divide operations (alldiv), and all floating-point operations (allfp). These groups were chosen because the corresponding instructions share many hardware resources.

7.2.1 Coverage

The numbers under the groups in Figure 7 represent the percent of execution-unit area covered by that group of instructions. Recall from Figure 2 that there was potentially 16% coverage via microarchitectural redundancy. In comparison to microarchitectural redundancy's low coverage, each of the floating-point multiply, int/simd shuffle, and division hardware represent individually between 5 and 7 percent of the execution-unit area. Note that integer-divide alone does not cover any area because it shares resources with floating-point divide. Allowing core salvaging in the presence of completely defective floating-point units provides 34.7% coverage of the execution-unit areas. The values represent a subset of the potential 83% coverage of execution-unit area shown in Figure 5.

7.2.2 Throughput

The throughput values in Figure 7 (A) assume that core salvaging is enabled, and processors fall-back to core disabling for an interval of 150,000 cycles if more than 2 migrations occur during a rolling 40,000-cycle window.

As expected, for most of the potential defects, we salvage enough performance from the defective core to exceed core disabling (i.e., exceeds relative to 7 defect-free cores). Average throughput across all of the defect cases is between 5% and 7% better than core disabling for all workloads except multimedia. For multimedia, throughput is 3% better than core disabling. Throughput is particularly good in the case of defective integer and simd shuffle (8%, 8%, 9%, 11% and 5% for spec2k-int, spec2k-fp, spec06, server, and multimedia), floating-point ROM (10%, 8%, 10%, 13%, and 8% for spec2k-int, spec2k-fp, spec06, server, and multimedia), or simd-shifter (10%, 9%, 10%, 13% and 8% for spec2k-int, spec2k-fp, spec06, server, and multimedia). These cases approach the throughput of a defect-free 8-core die and exceed core disabling for all workloads.

One case that does not perform well is a core with a defective integer multiplier. Throughput is 2%-3% below that of a 7 core die for both server and multimedia workloads, and more than 5% below that of a defect-free die for all workloads. Based on this throughput, and because integer multiply is used in a large fraction of benchmarks as mentioned in Section 7.1, integer multiply may *not* be a good candidate for core salvaging.

Of the three cases that cover the largest amount of area, i,si-shuf, alldiv, and allfp, the first two have good throughput that exceeds that of core disabling for all workloads. (Multimedia barely outperforms on alldiv because of the large number of floating-point divides). Allfp, which covers over a third of the execution area, does not fare as well because so many resources are disabled, but throughput is still higher than core disabling for spec2k-int and server workloads.

7.2.3 Optimizing the fall-back interval

This subsection shows the impact of fine-tuning the interval for which core-salvaging falls back to core disabling in the presence of too many migrations. Too small of an interval can result in thrashing due to repeated migrations, but too long of an interval can waste execution potential on the defective core. Figure 7 (B) shows results for each of our workloads for various fall-back intervals for the case of one core which cannot execute any divides (alldiv). We choose the alldiv case because it is one of our worst performers, and the aim of falling back to core disabling is to mitigate throughput loss.

Unlike the previous Figure 7(A), the top bar represents mean throughput compared to 8 defect-free cores, and the bottom bar represents a workload's worst-performing benchmark's throughput compared to 8 defect-free cores. Also unlike the previous figure, each group of bars represents one workload, and the bars within the group each interval size. The interval sizes are defined by two numbers in the format X/Y. X is the number of cycles for which we fall back to core disabling, and Y is the size of a cycle window during which we allow 2 migrations to occur before we fall back to core disabling. The cases are shown in the figure (a-d).

The first two bars for each workload vary the tolerance window for migrations before falling back to core disabling, specifically a) tolerates two migrations in 60,000 cycles and b) tolerates two migrations in 40,000 cycles. There is little difference between those cases. In other experiments (not shown), we found throughput is generally insensitive to the size of the tolerance window for values less than 100,000 cycles or to the number of migrations allowed in the tolerance window, for values less than 5.

The last three bars present increasing fall-back windows from 150,000 cycles to 250,000 cycles. Increasing the size of the fall back window improves worst-case throughput slightly for spec2k-int, spec2k-fp, and spec06, but degrades worst-case throughput for server and multimedia. The degradation for window sizes above 150,000 cycles is substantial for multimedia, indicating that the long fall-back to core disabling is denying useful opportunity to execute on the salvaged core. Because of that behavior, and because the impact of fall-back window size on average-case throughput is minimal, we choose a fall-back window of 150,000 cycles as our best case.

7.3 Hybrid Hardware Redundancy

In this section, we show results for core salvaging based on a hybrid of artificial microarchitectural redundancy and architectural redundancy, as discussed in Section 5. When one core is using a small secondary structure in place of a primary structure, we expect throughput to fall between that of a defect-free processor and one using core disabling. For the results shown, we again use our worst-case homogeneous workloads. Because workloads are homogeneous, the scheduling optimizations discussed in Section 5.1 would not be applied.

Figure 8 shows our results. We evaluate three scenarios which deviate from the baseline core given in Table 3: 1) a core relies only on a bimodal branch direction predictor, 2) a core relies on an

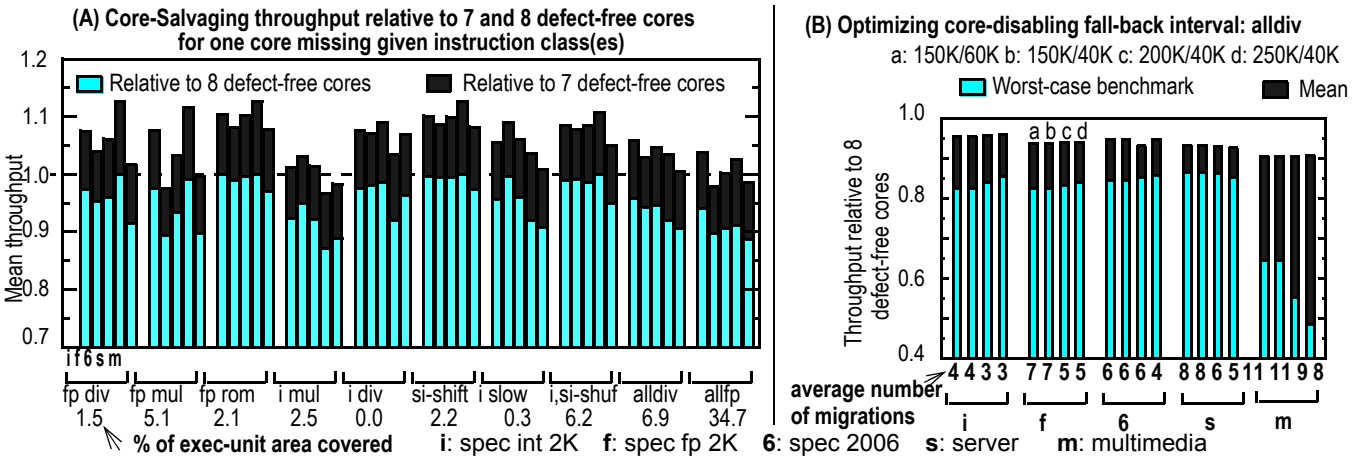

Figure 7: Core salvaging throughput.

10-entry instruction decode queue, and 3) a core relies on small secondary load and store buffers. Each scenario is a group of bars, and each bar represents one of our workloads.

The numbers under the scenarios in the figure represent the fraction of core area covered by allowing core salvaging (not including cache and TLB array area, as discussed earlier in Section 3.3). The coverage for these sample structures compares favorably to the potential coverage for microarchitectural salvaging, as shown earlier in Figure 4. Figure 4 showed microarchitectural salvaging covering a total of 10.4% for all small arrays, execution units, and simple decoders, while these three example structures alone add up to 12% coverage.

For the processor relying on a smaller instruction decode queue, all workloads outperform core disabling substantially, likely because pressure on the decode queue is low during most execution. The results for the small load and store buffers are mixed, but all but spec2k-fp at least outperform core disabling. Spec2k-fp and multimedia are both memory intensive, which likely causes their relatively poor performance with the secondary load and store buffers.

Finally, salvaging cores with defective branch predictors provides a model example where core salvaging is beneficial on all workloads. For the processor relying on only the bimodal branch predictor, no workload loses more than 1% average performance compared to 8 defect-free cores, and all have 9% or 10% higher throughput than core disabling.

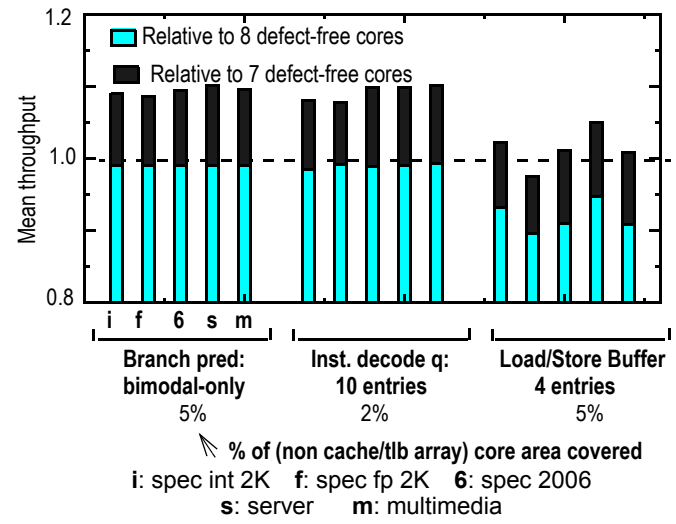

Figure 8: Hybrid core salvaging throughput.

8 CONCLUSION

The incidence of hard errors in CPUs is a challenge for future multicore designs due to increasing total core area. Even if the location and nature of hard errors are known a priori, either at manufacture-time or in the field, cores with such errors must be disabled in the absence of hard-error tolerance. While caches, with their regular and repetitive structures, are easily covered against hard errors by providing spare arrays or spare lines, structures within a core are neither as regular nor as repetitive. Previous work has proposed microarchitectural core salvaging to exploit structural redundancy within a core and maintain functionality in the presence of hard errors. Unfortunately microarchitectural salvaging introduces core complexity and may provide only limited coverage of core area against hard errors due to a lack of natural redundancy in the core. We showed that microarchitectural redundancy may have the potential to cover only about 10% of the vulnerable core area.

This paper makes a case for architectural core salvaging. We observe that even if some individual cores cannot execute certain operations, a CPU die can be instruction-set-architecture (ISA) compliant, that is execute all of the instructions required by its ISA, by exploiting natural cross-core redundancy. We propose using hardware to migrate offending threads to another core that can execute the operation. Architectural core salvaging can cover a large core area against faults, and be implemented by leveraging known techniques that minimize changes to the microarchitecture. We show it is possible to optimize architectural core salvaging such that the performance on a faulty die approaches that of a fault-free die--assuring significantly better performance than core disabling for many workloads and no worse performance than core disabling for the remainder.

We showed that architectural core salvaging has the potential to cover 86% of execution-unit area, and we showed proofs-of-concept covering 46% of the execution-unit area. That 46% of execution-unit area equates to 9% of vulnerable core area. We also showed proofs-of-concept for hybrid redundancy techniques that cover an additional 12% of core area. Thus, the total coverage of our examples is 21% of core area.

In contrast, microarchitectural salvaging potentially covered only 10.4% of vulnerable core area across *all* of the execution units, decoders, and small arrays. If the full potential coverage of execution units was utilized in addition to our hybrid coverage, architectural salvaging would cover up to 30% of vulnerable core area. Thus architectural core salvaging can provide substantially more coverage than microarchitectural salvaging without the performance loss of core disabling, while incurring minimal core changes and maintaining throughput near that of a fault-free CPU.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the work of Omer Khan whomade significant improvements to the Asim model while interning with the AMI group within SPEARS at Intel Massachusetts.

REFERENCES

[1] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO 32)*, Nov. 1999.

[2] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *International Conference on Dependable Systems and Networks (DSN2004)*, pages 51–60, June 2004.

[3] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO 38)*, pages 197–208, Nov. 2005.

[4] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, 2000.

[5] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S.-L. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka, S. Rusu, and D. Srivastava. The 65nm 16mb on-die l3 cache for a dual core multi-threaded xeon processor. In *2006 Symposium on VLSI Circuits*, pages 126–127, Feb. 2006.

[6] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO 40)*, pages 97–108, Dec. 2007.

[7] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patel, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. In *IEEE Computer 0018-9162:68-76*, pages 68–76, Feb. 2002.

[8] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A sub-lw to 2w low-power IA processor formobile internet devices and ultra-mobile PCs in 45nm hi-k metal gate CMOS. In *2008 IEEE International Solid-State Circuits Conference*, Feb. 2008.

[9] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. In *Proceedings of Seventeenth Symposium of IEEE Hot Chips*, Aug. 2005.

[10] S. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. In *Intel Technology Journal Q1 2001*, Q1 2001.

[11] Intel Corporation. *First Details on a Future Intel Design Codenamed Larrabee*. http://www.intel.com/pressroom/archive/releases/20080804fact.htm, Aug. 2008.

[12] Intel Corporation. *Intel Core 2 Duo Processor and Intel Core 2 Extreme Processor on 45-nm Process for Platforms Based on Mobile Intel 965 Express Chipset Family*. ftp://download.intel.com/design/mobile/datashts/31891401.pdf, Jan. 2008.

[13] Intel Corporation. *Intel Corporation's Multicore Architecture Briefing*. http://www.intel.com/pressroom/archive/releases/20080317fact.htm, Mar. 2008.

[14] D. A. Jimenez. Reconsidering complex branch predictors. In *Ninth International Symposium on High Performance Computer Architecture (HPCA)*, pages 43–52, Feb. 2003.

[15] R. Joseph. Exploring salvage techniques for multi-core architectures. In *Workshop on High Performance Computing Reliability Issues (HPCRI) 2005*, Feb. 2005.

[16] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *International Conference on Dependable Systems and Networks (DSN2008)*, pages 80–89, June 2008.

[17] M. D. Powell, A. Biswas, J. Emer, S. S. Mukherjee, B. R. Sheikh, and S. Yardi. CAMP: A technique to estimate per-structure power at run-time using a few simple parameters. In *Fifteenth International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2009.

[18] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 2008 International Conference on Parallel Architectures and Compiliation*, pages 43–51, Oct. 2008.

[19] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32st International Symposium on Computer Architecture (ISCA 32)*, pages 160–171, June 2005.

[20] E. Schuchman and T. N. Vijaykumar. Blackjack: Hard error detection with redundant threads on smt. In *International Conference on Dependable Systems and Networks (DSN2007)*, pages 327–337, June 2007.

[21] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *International Conference on Computer Design (ICCD)*, 2003.

[22] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Workshop on Silicon Errors in Logic - System Effects (SELSE-3)*, Apr. 2007.

[23] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32st International Symposium on Computer Architecture (ISCA 32)*, June 2005.

[24] The Standard Performance Evaluation Corporation. Spec CPU2000 suite. http://www.specbench.org/osg/cpu2000/.

[25] The Standard Performance Evaluation Corporation. Spec CPU2006 suite. http://www.specbench.org/osg/cpu2006/.

[26] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, HP Laboratories, Palo Alto, 2008.

[27] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1523–1529, 2002.