

Exploring Salvage Techniques for Multi-core Architectures

Russ Joseph

Department of Electrical Engineering and Computer Science

Northwestern University

Evanston, IL 60208

rjoseph@ece.northwestern.edu

Abstract

As process technology scales, both fabrication induced and in-operation hard faults will become more prevalent, limiting yield and effective product lifetime. The simultaneous emergence of chip multiprocessors (CMPs) and revitalization of machine virtualization offers several opportunities for hard failure tolerance. In this paper, we provide preliminary analysis of methods for lifetime recoverability on CMPs which contain partially functional execution cores. Specifically, we examine how processor virtualization can help a CMP architecture overcome faults by migrating computation or virtualizing functionality which cannot be supported by the hardware as a result of failure.

1 Introduction

In future generation microprocessors, reliability and defect tolerance will emerge as such prominent factors that architects will have to develop innovative solutions to address hard faults. First, malfunctions in-field threaten the functional lifetime of computer hardware. Second, manufacturing defects will become an increasingly vexing problem for chip makers due to several low level consequences of technology scaling [14].

Discarding a chip with a few faulty sub-circuits, may no longer be an economically viable option. Hence, it will become increasingly important to develop architectures and design methodologies that identify faulty logic and re-adjust the microarchitecture to use other execution structures in lieu of the faulty logic. Salvage approaches hold promise for curbing the impact of both defect related faults as well as in-field failures.

In chip multiprocessors (CMP), core-level fault detection and recovery would appear to be a natural granularity at which to work. Faulty cores can be identified, and spare cores can be swapped in to replace them. This however, can be rather hardware inefficient since the spare cores remain idle until they are called into service. Furthermore, a single fault anywhere in the core demands that the entire core be abandoned.

A second, alternative is to identify intra-core failures at the component level, and provide duplicate execution hardware to compensate in case of failure. This technique has the potential to be significantly more efficient. However, both of these approaches ignore that fact that homogenous CMPs already have inherent functional resource duplication across cores. In addition, some intra core execution resources may be extended to provide functionally equivalent results.

In this work, we explore core salvage techniques which rely on systems software and modest hardware support to make use of partially damaged cores. This software acts as a virtualizing agent which can create the abstraction of logical processor cores capable of executing any instruction. In reality, these logical cores are constructed out of partially damaged physical cores. Via instruction emulation and thread migration, the reliability management software can hide failures from application programs without incurring the overhead of explicitly replicating execution resources or using spare cores. We outline the implementation of a simple core salvaging agent and evaluate its behavior on a high-performance multi-core architecture.

The remainder of this paper is organized as follows: Section 2 examines prior related work in architectural fault detection and redundancy schemes. In Section 3, we introduce techniques for salvaging cores. We go on to describe our experimental methodology in Section 4. In Section 5, we report on the efficacy of the proposed solutions, and we conclude in Section 6.

2 Related Work

In response to the growing threat of hard-failure architects have proposed innovative schemes to detect hardware faults. For example, recent work by Bower et al. has examined an on-line mechanism to detect hard faults [4]. This work assumes use of a hardware checker ([1]) that identifies errors and increments a corresponding counter. Initially, these mismatches are assumed to be the outcome of soft-errors. When the counter reaches a threshold value, however, the hardware deduces that the responsible unit has a hard fault. Previous work has also examined detection and recovery mechanisms for soft errors [7, 18].

Once these faulty structures have been detected, the next step is to disable them and find alternative execution resources. Schuchman and Vijaykumar introduced a microarchitecture that used design for test (DFT) techniques to identify manufacturing defects, and map out faulty portions of the pipeline. Recent work has also demonstrated that array structures can be repaired via use of spare entries [3]. Finally, course grain execution resource duplication has been examined as a means to extend processor lifetime characteristics [17]

We identify two important limitations in prior work:

- First, a focus on single core processors limits the scope from which the hardware can apply redundancy to overcome failure.
- In addition, previous literature has largely assumed that hardware is responsible for salvaging cores which have failure, precluding software intervention.

In this work, we note that extending *within core* redundancy techniques to CMP architectures alone is rather pessimistic because it prevents redundancy sharing across cores. This may mean that individual cores may be over-provisioned for reliability, likely incurring some overhead in terms of area, energy, or design time.

The second trend, a hardware centric focus, may be limiting if the failing resource is used relatively infrequently by the currently executing thread. In this case, software can provide support either through migration or emulation of the missing functionality. Code emulation allows system software to mask the failure from the application code by reusing existing functionality within the core. Migration, on the other hand, makes redundancy sharing across cores possible.

3 Mechanisms for Core Salvage

While built-in self repair (BISR) mechanisms have been proposed for regular structures such as caches, tables, and circular queues [3], irregular logic, for example in an floating point execution unit, represents a greater challenge. While array structures can be over-provisioned with spare entries that can be used to replace faulty ones, it is not clear how this can be done in general for arbitrary logic. One alternative for course-grain redundancy is to duplicate any irregular units needed for program execution. However, this simplistic approach will likely incur significant overhead.

In this paper, we explore core salvage techniques for CMP architectures. Our approach leverages system software to effectively reuse *local* execution resources to replace lost functionality as well share execution resources *globally* across cores. This can improve the useful life of a core beyond the failure of a component or group of components or allow for

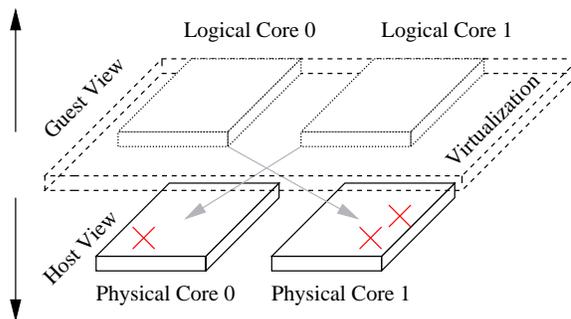


Figure 1: Core virtualization can mask hard failures and provide a substrate for thread migration. The indirection is provided by logical cores which appear no different to software than the fully functional cores it expects.

defect tolerance without undue hardware overhead. Moreover, we focus on irregular logic such as execution units due to the limited applicability of fine-grain BISR techniques to this type of logic. However, this discussion and analysis does not preclude use of fine-grain BISR techniques on regular array structures. These approaches are in fact orthogonal.

3.1 Core Virtualization

In this work, we examine *core virtualization* as a strategy for salvaging partially functional cores. In essence, virtualizing software creates the abstraction of logical processor cores which have the full functionality expected by application programs. However, the programs actually run on physical cores which may or may not have full hardware support for all instructions. With minimal hardware support, the virtualizing software can intercept instructions which cannot be directly supported due to failure and find an appropriate way to compute the needed results.

The chief benefit of this approach is that no application software modification is necessary. Virtualization masks the failure from the high-level code. Figure 1 illustrates the indirection provided by the virtualization. The virtualizing software could be implemented in the operating system or as firmware in a co-designed virtual machine monitor (VMM)[5, 6].

The obvious advantage to core virtualization in the operating system is that the salvage techniques can be directly integrated with scheduling, process accounting, and other operating system activities. This could help reduce overhead and allow the salvage routines to leverage higher level operating systems knowledge to improve throughput or guarantee fairness. On the other hand, the operating system would have to be modified to support new implementations of the same ISA.

An alternative is to implement the virtualizing software as a firmware VMM. This removes the need for any operating system upgrades or patches as new processor designs are introduced. The operating system would run as a guest, on top of the system-level VMM [16]. The main drawbacks are that the VMM would be without use of high-level operating information about the applications and a lost ability to coalesce and integrate functionality. In this paper, we assume a stand-alone VMM for reliability management. We plan to investigate integration with the OS in future work.

3.2 Migration and Emulation

When an application requires execution resources which are not present due to failure, the VMM intercepts the problematic instructions and emulates the missing functionality. This allows computation to resume, effectively masking the failure. Emulation as a means to extend functionality is a fairly common practice in some domains, especially the embedded programming world. Some popular, low-power microprocessors do not have native support for floating-point computation. Software packages, such as the GNU `soft-fp` library, provide support for applications that perform floating point calculations. Clearly emulation incurs a penalty over native hardware execution, however, the overall performance cost may be reasonable if the failing resource is used infrequently. For example, a failed multimedia extension unit might not have a negative impact on traditional integer/floating-point applications such as those found in SPEC.

The type of emulation support required depends largely on the failure modes. In some instances, it may be impossible to recover from failures because critical resources are unavailable. In this paper, we assume a clustered architecture similar to the Alpha 21264 [9]. Specifically, we assume that there are three clusters:

- Complex Integer Cluster: load/stores and any integer instruction (including integer multiplies)
- Simple Integer Cluster: load/stores and any simple (e.g. integer ALU operation: add, subtract, logical)
- Floating Point Cluster: all floating point instructions

Figure 2 presents a pictorial view of the execution pipelines assumed in this paper. We consider these clusters to be the smallest granularity of failure in this paper. For example, we do not separately examine failure of floating point add and multiply units. In addition, we also assume the integer execution units may access floating point registers via load/store operations, even after floating point execution units have failed. This allows the VMM to retrieve and writeback

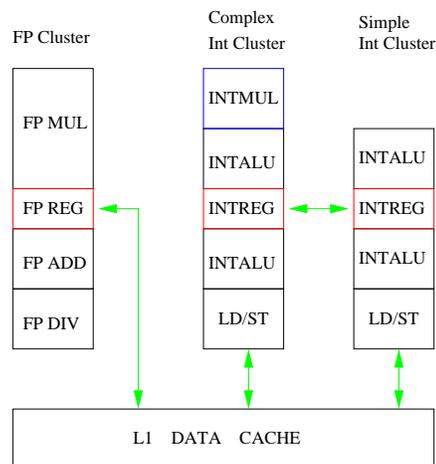


Figure 2: Cluster organization assumed in this paper.

FP registers when it emulates instructions. In this work, we look at three specific failure combinations: (a) Complex Integer, (b) Floating Point, (c) Complex Integer and Floating Point. At minimum, this requires emulation support for complex integer instructions (using the simple cluster) and floating point instructions (we again assume use of simple cluster, to permit simultaneous Complex Int and FP failures). Note that because the complex cluster operations are a superset of the simple cluster operations, no emulation is required when the simple cluster fails. Natural, operational redundancy at the cluster level guarantees that execution resources will be available. While execution on this core would incur some performance loss due to the loss of issue bandwidth, this scenario can be considered a special case of a heterogeneous core architecture [11, 12], hence we do not examine it here. Finally, we assume that at least one integer unit is available, since both integer and floating-point benchmarks make heavy use of integer operations.

If the emulation overhead for an application becomes significant, the VMM has an alternative. It may choose to migrate computation to another core, possibly an idle one. If all the cores are occupied, the VMM would have to swap threads between a damaged and a fully functional one. Through virtualization, the applications and operating system do not witness the architected effects of the migration. In particular, we assume that VMM intercepts all reads/writes to low-level status registers that might identify the physical ID of a core.

Due to effects that execution has on architected state (registers and cached instructions/data) and non-architected state (predictor tables and meta data), we must also consider the cost of migration as implemented by the VMM. Registers need to be explicitly copied to and from shared memory for process migration. We assume that TLBs use ASID bits to avoid flushes on every context switch and that caches operate on physical addresses. After migration, the data/instructions

will also migrate on access via cache coherency protocols. These access will be subject to writeback penalties and standard memory hierarchy penalties. Finally, predictive state does not need to be explicitly propagated for correctness. In this study, we assume that the VMM does not transfer branch predictor information between cores. Due to the cumulative costs, migration must be performed judiciously to limit the effects of TLB/cache misses and branch mispredictions as threads hop from core to core.

Finally, we note that the VMM must strike a balance between throughput (overall performance) and fairness (performance on individual cores) [10, 8]. When a single core has an execution unit failure, the VMM may choose to: (a) migrate an application that would be less impacted by the failure to the damaged core, (b) distribute the performance impact by rotating cores to the failing processor, or (c) not migrate at all. Ideally, the impact of the failure can be minimized by finding an application which does not require the failed resource. For example, if a floating-point unit fails, migrating an integer application to that core would probably have little to no impact on overall performance. In general, this may not be a feasible solution, and some compromise between fairness and global performance would have to be reached. In Section 5, we investigate a simple round robin policy for migration. We leave a more thorough investigation of more advanced selective migration policies for future work.

3.3 Hardware Support

To support an efficient implementation of virtualized core salvaging, the execution hardware must have some important underlying technologies. We note however, that most of these techniques have already been proposed to support reliability and defect tolerance, and are purported to have reasonable hardware overhead.

The most important support is the ability to detect and recover from hard-failures. Although this is clearly a non-trivial task, recent work suggests that there are realistic ways to extend soft-error detection to uncover and isolate hard faults [4]. Once detected, the faulty structures can be decommissioned or “mapped out”, preventing future instructions from using them. A related issue is defect tolerance for manufacturing induced failure. Recent work has also examined ways to apply fine-grain microarchitectural design for test (DFT) and map out functionality [13]. In field-failure recovery and defect tolerance can be regarded as different aspects of the same problem with the key difference being when the faults are detected: on-line vs. off-line. In this respect salvage can be applied in either case, with the aim of providing extended usability or reducing the need for structural duplication [17].

In addition to map out capabilities which would prevent an instruction from using a faulty resource, the VMM needs some hardware support to let it intercept instructions which

need emulation support. This can be provided by a trap into the VMM. One possibility is detection logic in decode stage which can be configured to mark a complex integer or floating point instruction and raise an exception if the instruction is discovered to be non-speculative.

4 Methodology

To evaluate the efficacy of firmware directed core salvaging, we model a high performance multi-core processor executing multiprogrammed workloads. Rather than directly implementing a VMM, we modified our detailed cycle level simulator to mimic the behavior of the VMM, including performance overhead for instruction emulation and thread migration. In particular, we assume a 50 cycle overhead for emulation of floating point add instructions, 100 cycles for integer multiplies, 150 cycles for floating point multiplies, and 300 cycles for floating point divides. We are currently developing a prototype VMM that we plan to use to get a better understanding of the efficiency of core salvage. Our prototype VMM will include true emulation of these instructions.

4.1 Processor Model

Our experiments model the performance of a 4-core homogenous chip multiprocessor for a 90nm process. Each core of the processor is comparable to an Alpha 21264 (EV6) scaled to current technology [9]. The cores in the processor have private L1 data and instruction caches, but share an L2 which is accessed via a global L2 bus. Table 1 summarizes our base processor model. Our simulation infrastructure is based on the M5 Stand-Alone Execution simulator [2].

Single Core	
Fetch/Decode Width	4 inst
Issue Width	6 inst, out-of-order
IQ/LSQ/ROB	32/40/80 entries
Functional Units	4 IntALU, 1 IntMult 1 FPALU, 1 FPMul, 1 FPDIV 2 MemPorts
L1 Inst Cache	64KB 2-way 64B blocks
L1 Data Cache	64KB 2-way 64B blocks 3 cycle load hit
Chip Multiprocessor	
Cores	4
L2	8MB 8-way shared 128B blocks
Off-chip memory latency	200 cycles
VDD	1.2V
Clock Rate	2.5GHz
Feature Size	90nm

Table 1: Processor Parameters

4.2 Workloads

To evaluate the efficacy of our wear management approach, we use several workload mixes that showcase a variety of resource utilization patterns which would be likely in a multiprogrammed environment. Individual applications are taken from the SPEC CPU2000 benchmark suite. To reduce the total number of simulations, we identify eight SPEC applications, four integer and four floating point benchmarks and then group them into eight mix sets.

Table 2 shows the workload groupings used in our experiments. To isolate representative simulation windows, we use SimPoint to identify early, representative execution simulation points [15] for all benchmarks.

Workload	Benchmarks
mix-1	eon, mesa, crafty, bzip2
mix-2	equake, swim, twolf, ammp
mix-3	eon, twolf, crafty, bzip2
mix-4	equake, swim, mesa, ammp
mix-5	eon, mesa, twolf, ammp
mix-6	crafty, bzip2, equake, swim
mix-7	eon, mesa, equake, swim
mix-8	crafty, bzip2, twolf, ammp

Table 2: Workloads used in this paper.

5 Results

To evaluate the impact of firmware core salvaging on performance, we applied the methodology described in previous section and examined three fault combinations: a single faulty complex integer cluster, a single faulty floating point cluster, and a combined failure of a complex integer cluster on one core and failed floating point cluster on a separate core. Figure 3 shows the relative slowdowns for all of our mix sets.

In general, failure of the complex integer cluster has the least impact on performance (on average 9.2% performance loss). The floating point and combined failure modes lead to performance losses of 11.4% and 18.8%, respectively. As expected, different workloads have differing degrees of sensitivity to the various failures. For example, the FP failures have significantly more impact on workloads that are dominated by floating point applications. In addition to the expected impact that complex integer failures have on integer workloads, they in many cases have a large impact on floating point applications as well. For instance, mix-2 suffers a 19% percent performance loss when a complex integer cluster fails. In this case, failure of the complex integer cluster also makes a load/store unit unavailable. The benchmarks in mix-2 (equake, swim, twolf, and ammp) are memory stressors.

A surprising result is that under floating point cluster failures, there is still a significant performance hit for integer

dominated workloads. This is not a result of emulation overhead, rather it represents migration overhead. Under the simple strategy used in this paper, cores migrate in a round robin fashion, regardless of the dominant mode of computation and the existing failure. A simple modification to this policy, restricting migration when there are no clear benefits, would alleviate this problem. This could be done by constructing a cost function for migration and weighing it against the the expected performance if migration is suspended.

While these performance losses are far from negligible, they are clearly more palatable than having a failure exposed, rendering the chip useless. In addition, the low hardware overhead of firmware core salvaging also makes it an attractive alternative to explicit duplication of pipeline resources.

6 Conclusion

In this paper, we introduced firmware based core salvaging as a means to recover functionality in partially damaged CMPs. By virtualizing physical cores, system software can emulate instructions that cannot execute due to faulty hardware. In addition, thread migration provides a means to achieve fairness. Rather than saddling a specific, unlucky thread with a faulty core, the virtualizing agent can take advantage of natural functional resource duplication to spread the impact of the failure across cores. In all, core salvage is a low-cost alternative to course grain structural duplication.

References

- [1] T. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual Symposium on Microarchitecture (MICRO-32)*, November 1999.
- [2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using m5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2003.
- [3] F. A. Bower, P. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2004.
- [4] F. A. Bower, P. Shealy, S. Ozev, and D. J. Sorin. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO-38)*, November 2005.
- [5] A. Dhodapkar and J. E. Smith. Saving and restoring implementation contexts with co-designed virtual machines. In *Workshop on Complexity Effective Design at ISCA-28*, June 2001.

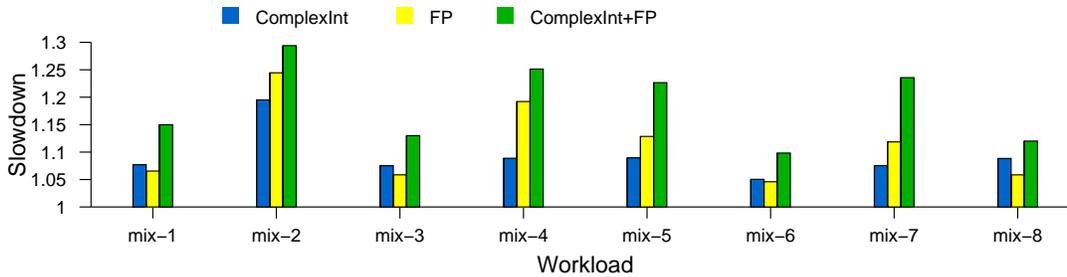


Figure 3: Relative slowdown under core salvage for various execution unit failures. We assume exactly one failure over the entire chip for ComplexInt and FP execution unit failures. For ComplexInt+FP, we assume a single failure of each on different physical cores.

[6] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[7] M. Goamaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of 30th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[8] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *The 35th International Symposium on Microarchitecture (MICRO-35)*, 2002.

[9] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[10] M. F. K. Luo, J. Gummaraju. Balancing throughput and fairness in smt processors. In *International Symposium on Performance Analysis of Systems and Software*, Jan. 2001.

[11] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, December 2003.

[12] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, June 2004.

[13] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.

[14] Semiconductor Industry Association. International Technology Roadmap for Semiconductors, 2003. <http://public.itrs.net/Files/2003ITRS/Home.htm>.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002.

[16] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman Publishers, 2005.

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

[18] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery via simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, May 2002.