Appears in the International Conference on Parallel Architectures and Compilation Techniques (PACT)
Toronto, Canada, October, 2008

Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults

Bogdan F. Romanescu and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
Durham, NC
{bfr2, sorin}@ece.duke.edu

ABSTRACT

To improve the lifetime performance of a multicore chip with simple cores, we propose the Core Cannibalization Architecture (CCA). A chip with CCA provisions a fraction of the cores as cannibalizable cores (CCs). In the absence of hard faults, the CCs function just like normal cores. In the presence of hard faults, the CCs can be cannibalized for spare parts at the granularity of pipeline stages. We have designed and laid out CCA chips composed of multiple OpenRISC 1200 cores. Our results show that CCA improves the chips' lifetime performances, compared to chips without CCA.

Categories and Subject Descriptors
B.8.1 Reliability, Testing, and Fault-Tolerance, C.1.0 Processor Architectures, C.4 Performance of Systems

General Terms
reliability, performance

Keywords
multicore, lifetime performance, fault tolerance, reliability

1. INTRODUCTION

Technology trends are leading to an increasing likelihood of hard (permanent) faults in processors [21]. Smaller transistors and wires are more susceptible to hard faults. Furthermore, as we continue to add more transistors and wires to chips, there are more opportunities for hard faults. These hard faults may be introduced either during fabrication or in the field.

One traditional approach to this problem is to provision spare components. Unfortunately, spare components (either cold or hot) consume hardware resources that provide no performance benefit during fault-free operation. If we provision spares for all components, then we achieve approximately half the fault-free performance of an equal-area chip without spares.

The goal of our work is to tolerate hard faults in multicore processors without sacrificing hardware for dedicated spare components. There are two aspects to multicore processors that distinguish the issue of self-repair from the case for single-core processors. First, power and thermal constraints motivate the use of simple, in-order cores, perhaps in conjunction with one or two superscalar cores. Examples of multicore chips with simple, narrow cores include the UltraSPARC T1 [11] and T2 [16], Cray MTA [4], empowerTel MXP processor [7], Renesas SH-2A-Dual [23], and Cisco Silicon Packet Processor [5]. Unfortunately, simple cores have little intra-core redundancy of the kind that has been leveraged by superscalar cores to provide self-repair [3, 17, 22]. Just one hard fault in the lone ALU or instruction decoder renders a simple core useless, even if the entire rest of the core is fault-free. The second aspect of self-repair that is distinct to multicore processors is the opportunity to use resources from fault-free cores. One existing self-repair scheme is to shut down any core with one or more hard faults and just use the remaining cores. This core shutdown (CS) solution, however, wastes much fault-free circuitry.

In this paper, we propose the *Core Cannibalization Architecture (CCA)*, which is the first design of a low-cost and efficient self-repair mechanism for multicore processors with simple cores. The key idea is that one or more cores can be cannibalized for spare parts, where parts are considered to be pipeline stages. The ability to use stages from other cores introduces some slight performance overhead, but this overhead is outweighed by the improvement in lifetime chip performance in the presence of multiple hard faults. Furthermore, CCA provides an even larger benefit for multicore chips that use cores in a triple modular redundancy (TMR) or dual modular redundancy (DMR) configuration, such as Aggarwal et al.'s approach [1]. CCA enables more cores to be operational, which is crucial for providing enough cores for TMR or DMR.

We have developed several concrete implementations of CCA in the context of processors that consist of simple OpenRISC 1200 cores [12]. Our results show that, over the chip's lifetime, CCA achieves better performance than CS. After only 2 years, CCA chips outperform CS chips. Over a lifetime of 12 years, CCA achieves a 63% improvement in cumulative performance for 3-core chips and a 64% improvement for 4-core chips. Furthermore, if cores are used redundantly (e.g., TMR or DMR), then CCA's improvement is 70% for 3-core chips and 63% for 4-core chips.

In the rest of this paper, we describe CCA in general (Section 2) and present implementations of CCA (Section 3). We then evaluate CCA (Section 4) and compare it to prior research (Section 5).

2. OVERVIEW OF CORE CANNIBALIZATION

In this section, we first present our system model. Next, we discuss core shutdown (CS), since it is the natural design point against which to compare. We then provide a high-level description of CCA and explore the design space for CCA.

2.1. Core Model

In this work, we focus on simple, in-order cores with little redundancy. We present CCA in the context of 1-wide (scalar) cores, but CCA also applies to many cores that are wider but still have numerous single points of failure. There are many *k*-wide cores that cannot tolerate a fault by treating the core as being *k-1*-wide. For example, the Renesas SH-2A [23] is dual-issue, but it has only one shifter and one load/store unit. Any fault in either of those units renders the entire core unusable. Other simple cores are susceptible to numerous single faults (e.g., in the PC update logic) that affect all lanes of the processor. Many commercial cores fit our core model [11, 16, 4, 23, 5].

Our core model assumes that the core has mechanisms for detecting errors and diagnosing hard faults (i.e., identifying the locations of hard faults). Detection and diagnosis are orthogonal issues to self-repair, and acceptable schemes already exist, such as the built-in self-test (BIST) used by the BulletProof pipeline [18]. CCA may need a few more BIST test vectors than CS to distinguish faults that are in different pipeline stages and that would otherwise be exercised by the same test vector (which would be sufficient for CS).

2.2. Background: Core Shutdown

A multicore processor with C simple cores can tolerate hard faults in F ($F<C$) distinct cores by simply not using the faulty cores. A single fault in a core renders that core useless; additional faults in that core do not matter, since the core has already been shut off. We illustrate a 3-core processor with core shutdown in Figure 1. In the presence of three hard faults, one in each core, the processor achieves zero performance because none of its cores is operable. The performance of a chip with CS is proportional to C-F.

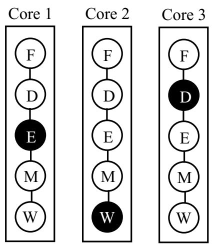


Figure 1. 3-core Chip with Core Shutdown. Core 1 has a faulty Execute stage, Core 2 has faulty Writeback stage, and Core 3 has faulty Decode stage.

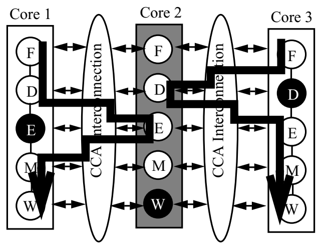


Figure 2. 3-core Chip with CCA. Core 1 and Core3 are NCs, and Core2 is a CC. Core1 has faulty Execute stage, Core3 has faulty Decode stage, and Core2 has faulty Writeback stage.

2.3. High-Level View of CCA

At a high level, a CCA processor resembles the system in Figure 2. There are some number of normal cores (NCs) that cannot be cannibalized as well as some number of cannibalizable cores (CCs). In this figure, there are three cores, one of which (Core2) is a CC. CCA enables Core1 to overcome a faulty Execute stage and Core3 to overcome a faulty Decode stage, by cannibalizing these stages from Core2. Despite the presence of three hard faults (including the fault in Core2's Writeback stage), Core1 and Core3 continue to function correctly. The performance of both cores is somewhat degraded, though, because of the delay in routing to and from the cannibalized stages. Comparing the chips in Figures 1 and 2, which both have three faults, we see that CS offers zero performance, yet CCA provides the performance of two slightly degraded cores.

In general, as the number of faults increases, CCA outperforms CS. For chips with zero or very few faults, a processor with CS will outperform CCA, because CCA's configurability logic introduces some performance overhead into the cores. This performance overhead is similar to that incurred by schemes that provide spare components. However, as the number of faults increases, CCA can tolerate more of them and provide a graceful performance degradation.

2.4. CCA Design Space

There are three important issues involved in the design of CCA or a scheme like it. After studying the first two issues, spare granularity and sharing policy, we have made fixed decisions for both of them. For the third issue, chip layout, we explore several options.

Spare Granularity. We decided to cannibalize spare components at the granularity of pipeline stages. The coarsest possible granularity is spare cores (i.e., CS), but coarse granularity implies that a single fault in a core renders the entire core useless. Finer granularity avoids wasting as much fault-free hardware, but it complicates the design, especially the routing to and from spare components. For example, one recent scheme for fine-grain redundancy [14] has an area overhead that is greater than 2x. We chose a granularity of pipeline stages because it offers a good balance between complexity and performance. We leave the search for an optimal granularity to future work.

Sharing Policy. Another issue to resolve is whether to allow multiple cores to simultaneously share a given component (pipeline stage). There are three options. First, at one extreme, a core with a faulty component of type Z "borrows" (time multiplexes) a component of type Z from a neighboring core that continues to function (is not cannibalized). A second option is to allow multiple cores to time multiplex a single cannibalized component. Both of these first two options introduce resource contention, require arbitration logic, and complicate pipeline control logic. For these reasons, we chose a third option, in which any given component can only be used by a single core.

Chip Layout. We must decide how to arrange the cores on the chip. We want to carefully arrange NCs and CCs to minimize the distance between NC stages and potential CC spare stages. We must also decide what fraction of the cores should be CCs.

3. CCA IMPLEMENTATIONS

In this section, we describe two concrete implementations of CCA. We first describe the baseline multicore processor (Section 3.1). We then present 3-core and 4-core CCA processors (Section 3.2 and Section 3.3). Lastly, we discuss how to extend CCA to chips with greater numbers of cores (Section 3.4).

Because CCA is concerned with low-level issues of chip layout and wire delay, evaluating it properly requires us to implement it at a low level of detail. We used our CAD tools, Synopsys Design Compiler and Cadence Silicon Ensemble, to floorplan and layout the chips described in this section, both with and without CCA. Our CAD tools use a proprietary 90nm standard cell library.

3.1. Baseline Multicore Processor

Our baseline processor consists of multiple OpenRISC 1200 (OR1200) cores [12]. Each OR1200 core is a scalar (1-wide), in-order, 32-bit core with 4 pipeline stages: Fetch, Decode, Execute, and Writeback. Each core has 32 registers and separate instruction and data caches. When this core is implemented in 90nm technology, our CAD tools estimate a clock frequency of roughly 400MHz.

An NC's use of a cannibalized CC stage introduces issues that are specific to that particular stage, so we now discuss the cannibalization of each stage.

Fetch. The Fetch stage accesses the I-cache. If an NC has to use a CC's Fetch stage, it also uses the CC's I-cache; the NC no longer uses its own I-cache.

Decode. In addition to decoding instructions, the Decode stage also reads registers and computes branch/jump target addresses. The register file is part of the Decode stage, so an NC that uses a CC's Decode stage also uses that CC's register

Figure 3. CCA3(2/1) layout. The shaded core, Core2, is a CC.

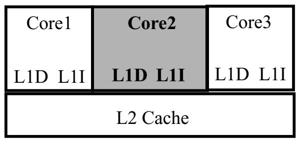


Table 1. Each Stage's Inputs and Outputs

stage	#input signals	#output signals
Fetch	56	65
Decode	38	115
Execute	110	61
Writeback	87	52

file. Thus, an NC that uses a CC's Decode stage must route back to the CC's register file during Writeback.

Execute. The Execute stage is where computations occur and where loads and stores access the D-cache. An NC that uses a CC's Execute stage also uses that CC's D-cache; the NC no longer uses its own D-cache.

Writeback. CCA does not require modifications to be made for Writeback, but it motivates a small change for register writing. Because the register writing logic is extremely small, it is preferable, in terms of area and performance, to simply replicate it (as a cold spare) in the original Writeback stage. Intuitively, forcing an NC to go to a CC for a tiny piece of logic is not efficient.

3.2. CCA3: 3-Core CCA Implementation

We first consider a 3-core chip that we refer to as CCA3(2/1). The "CCA3(2/1)" notation means that the chip has 3 cores, 2 of which are NCs and 1 of which is a CC. Our CCA3(2/1) implementation arranges the cores as shown in Figure 3, and we designate only the middle core, Core2, as a CC. By aligning the cores in the same orientation, we facilitate routing from an NC to a CC. By provisioning one CC, we obtain better chip performance than if we had implemented CCA3(1/2), which would have 1 NC and 2 CCs. With more than one CC, the fault-free performance of each core decreases, due to added wires and multiplexing, and the ability to tolerate more faults does not increase much. Note that, if a single fault occurs in either Core1 or Core3, it is preferable to just not use that core, rather than cannibalize Core2.

CCA3(2/1)'s reconfigurability requires some extra hardware and wires, similar to the overhead required to be able to use spare components. Each NC (Core1 and Core3) has multiplexors at the input to each stage that allow it to choose between signals from its own other stages (the majority of which are from the immediate predecessor stage) and those from the CC (Core2). Similarly, Core2 has multiplexors at the input to each stage that allow it to choose between signals from its other stages and signals from the two NCs. Table 1 shows the number of wires that are the inputs and outputs of each stage. These are the wires that cross between NCs and CCs and that must be multiplexed.

In CCA3(2/1)'s chip layout, the distance to route from Core1 or Core3 to Core2 and back is short. The cores are small, and

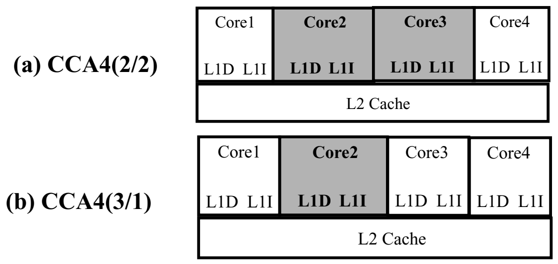


Figure 4. CCA4 Layouts. Shaded cores are CCs, and unshaded cores are NCs.

the distance each way is approximately 1mm in 90nm technology. Furthermore, because these simple cores are designed for power efficiency rather than for maximum clock frequency, we do not expect them to have very high clock rates. Thus, given a clock frequency in the 400 MHz range and such short wires, the penalty of routing to and from a cannibalized stage is a relatively small fraction of a clock period (as we show in Section 4.2.2). Rather than add wire delay pipe stages to avoid lengthening the clock period (which we consider for our 4-core implementations in Section 3.3), we simply slow the clock slightly. For chips with larger cores, adding wire delay pipe stages may be preferable.

One way to mitigate the impact of lengthening the clock period is to use clock borrowing [24]. Consider a fault in Core1. If Core1's normal clock period is T and its extra wire delay to and from Core2 is W, then a simplistic solution is to increase Core1's clock period to $T'=T+W$. Clock borrowing can mitigate this performance impact by time sharing W across its two neighboring stages [24]. By sharing this delay, we can reduce the clock period penalty to 1/3 of W, i.e., $T'=T + W/3$. As a concrete example, if Core1 has a 50ns clock period (T=50ns) when fault-free and W=15ns, then we can use time borrowing to achieve a clock cycle of T'=55ns. We borrow 5ns from both of the neighboring stages, pushing them from 50ns to 55ns. Thus, we have 65ns-10ns=55ns for the longer stage.

3.3. CCA4: 4-Core CCA Implementations

We illustrate two viable CCA4 arrangements in Figure 4. CCA4(3/1) chips are natural extensions of the CCA3(2/1) chip. The CCA4(2/2) chips, which have two cannibalizable cores, differ from the CCA4(3/1) chips in how they share stages. Core1 can use a stage from Core2 or Core3, Core2 and Core3 can use stages from each other, and Core4 can use a stage from Core3 or Core2. This sharing policy allows CCs to share with each other, and it allows the NCs to share from their more distant CCs.

An important distinction between CCA3 and CCA4 chips (of any kind) is that, in a CCA4 chip, an NC may have to borrow a stage from a CC that is not an immediate neighbor. For example, in Figure 4b, Core4 is approximately twice as far from a CC as Core3 is. Furthermore, as shown in Figure 4a, a given NC might have different distances to the two CCs (e.g., Core4's distance to Core2 and Core3).

The increase in distance from an NC to a CC may, for some core microarchitectures, discourage the simple approach of lengthening the clock period of an NC that is using a cannibalized stage. In Figure 4a, for example, there might be an unacceptable clock frequency penalty if we slow Core1 to accommodate using a cannibalized stage from Core3. Based on this clock penalty, we consider two approaches: the clock period lengthening we have already discussed and adding clock cycles to the pipeline. The first approach sacrifices clock frequency and the second approach sacrifices IPC. The preferred approach, in terms of overall performance, depends on the details of the core, so we discuss both here.

3.3.1. CCA4-clock

If the performance penalty of slowing the clock is preferable to adding pipeline stages, we can use what we refer to as the CCA4-clock design. The only new issue for CCA4-clock, with respect to CCA3, is that it is possible that we want to have different pipeline stages of the same CC operate at different frequencies. For example, in Figure 4b, if Core1 is using Core2's Decode stage and Core4 is using Core2's Execute stage, then we want Core2's Decode stage to be at a higher frequency than its Execute stage. This difference results from Core4 being further from the CC than Core1 is from the CC. Prior work has shown how to provide different clocks within a single core [10]. However, if such a solution is considered too costly, then Core2's clock frequency must be lowered to match the lowest frequency needed, such as the one imposed by Core4 in the example.

3.3.2. CCA4-Pipe

The CCA4-pipe design, like CCA3, assumes that routing from an NC to an immediately neighboring CC can be efficiently accommodated by lengthening the clock period of the NC and the CC. However, it allows routing from an NC to a CC that is not an immediate neighbor to take one additional cycle, and routing back from the CC to the NC takes another cycle. We do not lengthen the clock, because the wire and mux delays fit well within a cycle for a simple, relatively low-frequency core. To avoid adding too much complexity to the NC's control, we do not allow a single NC to borrow more than one stage that requires adding cycles.

When we add wire delay pipeline stages to a core's pipeline, we must add extra pipeline latches and solve four problems:

1) Conditional Branch Resolution. In the OR1200, the decision to take a branch is determined by a single BranchFlag signal that is continuously propagated from Execute back to Fetch. This BranchFlag is explicitly set/unset by instructions. Because the OR1200 has a single delay slot, the Fetch stage expects to see a BranchFlag signal that corresponds to the instruction that is exactly two instructions ahead of it in program order. However, adding cycles between Fetch and Execute can cause the BranchFlag signal seen by Fetch to be stale, because it corresponds to an instruction that is more than two ahead of it. To address this issue, we slightly modify the pipeline to predict that the stale BranchFlag value is the same as the value that would have been seen in the unmodified pipeline. We add a small amount of hardware to remember the program counter of a branch in case of a misprediction. If the prediction is correct, there is no penalty. A misprediction causes a penalty of two cycles.

2) Branch/Jump Target Computation. The target address is computed using a small piece of logic in the Decode stage, and having this unit close to the Fetch stage is critical to performance. We treat this logic separately from the rest of the Decode stage, and we consider it to be logically associated with Fetch. Thus, if there is a fault in the rest of the NC's Decode stage, it still uses its original target address logic. This design avoids penalties for jump address computation.

3) Operand Bypassing. When an NC uses a CC's Execute stage, there are some additional bypassing possibilities. The output of the CC's Execute stage may need to be bypassed to an instruction that is in the wire delay stage of the pipeline right before Execute. Instead of adding a bypass path, we simply latch this data and bypass it to this instruction when it reaches the usual place to receive bypassed data (i.e., when it reaches the Execute stage).

4) Pipeline Latch Hazards. The extra stages introduce two structural hazards for pipeline latches. First, if a cannibalized stage can incur an unexpected stall, then we must buffer this stage's inputs so they do not get overwritten. For the OR1200, Fetch and Execute require input buffering due to I-cache and D-cache misses, respectively. Second, if a cannibalized stage is upstream from (closer to Fetch than) a stage that can incur an unexpected stall, then the stall will reach the cannibalized stage late. To avoid overwriting the output of that stage, we buffer its output. For the OR1200, the Fetch and Decode stages require output buffering, because the Execute stage can stall on D-cache misses.

If the area costs of buffering are considered unacceptably high, it is possible to squash the pipeline to avoid the structural hazards. For example, a D-cache miss could trigger a squash of younger instructions. In our evaluation of CCA's area, we pessimistically assume the use of buffering rather than squashes, even though squashing on D-cache misses would have no IPC impact on the OR1200 (because, after the squash, the pipe would refill before the D-cache miss resolves).

3.4. CCA Chips with More Cores

Technological trends suggest that future chips will have far more than 3 or 4 cores. One feasible and straightforward way to apply CCA to chips with more cores is to design these chips as groups of CCA3 or CCA4 clusters. We leave for future work the exploration and evaluation of unclustered designs for chips with greater numbers of cores.

4. EVALUATION

We evaluated CCA to answer two questions. First, how much chip area overhead do our CCA implementations introduce? Second, how well do multicore processors consisting of CCA3 and CCA4 clusters perform, compared to CS processors?

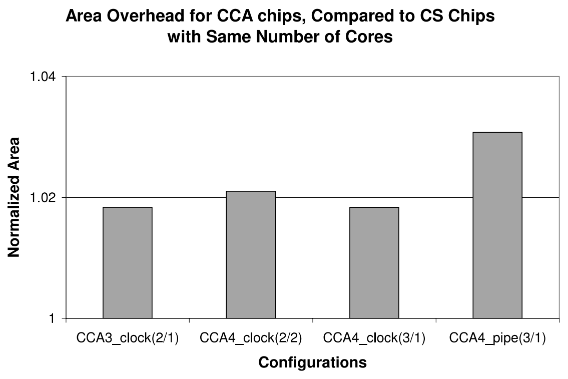


Figure 5. CCA Area Overhead

4.1. Area Overhead

CCA's area overhead is due to the logic and wiring that enable stages from CCs to be connected to NCs. In Figure 5, we plot the area overheads (compared to CS) for various CCA chip implementations in 90nm technology. These areas include the entire chip: cores and the L1I and L1D caches, which are both 8KB and 2-way set-associative. We consider all of the following CCA designs: CCA3(2/1), CCA4-clock(3/1), CCA4-pipe(3/1), and CCA4-clock(2/2).

We observe that no CCA chip has an area overhead greater than 3.5%. CCA3(2/1) incurs less than 2% overhead, which is a difference so small that it would require more than 50 cores on the chip—approximately 18 CCA3(2/1) clusters—before the additional area was equivalent to a single baseline core. The CCA4 overheads are comparable to the CCA3 overhead, except for CCA4-pipe, which requires some input/output buffering and modified control logic in the cores.

4.2. Lifetime Performance

The primary goal of CCA is to provide better lifetime chip performance than CS. We show in Section 4.2.4 that CCA achieves this goal, despite the small per-core performance overheads introduced by CCA. To better understand these results, we first present our fault model and then evaluate fault-free single core performance (for both NCs and CCs) and the performance of an NC using a cannibalized stage.

4.2.1. Fault Model

We consider only hard faults, and we choose fault rates for each pipeline stage that are based on prior work by both Blome et al. [2] and Srinivasan et al. [22]. Blome et al. decomposed the OR1200 core into 12 structures (e.g., fetch logic, ALU, load-store unit, etc.) and, for each structure, determined its mean time to failure in 90nm technology. Their analysis considered the utilization of each structure, and they studied faults due only to gate oxide breakdown. Thus, actual fault rates are expected to be greater [22], due to electromigration, NBTI, thermal stress, etc. Srinivasan et al. assume that fault rates adhere to a lognormal distribution with a variance of 0.5; the lognormal distribution is generally considered more realistic for hard faults due to wearout, because it captures the increasing rate of faults at the end of a chip's expected lifetime. The variance of 0.5 is a typical value for wearout phenomena [22]. By combining these two results, we can compute fault rates for each pipeline stage. We also consider faults in CCA-specific logic (including added latches and muxes), and we assume that these faults occur at a rate that is the average of the pipeline stage fault rates.

In our experiments, we consider these fault rates to be the nominal fault rates, and we also explore fault rates that are both more pessimistic (2x and 4x nominal) and less pessimistic (1/4x and 1/2x nominal). We assume that there are no faults present at time zero due to fabrication defects. The presence of fabrication defects would improve the relative lifetime performance of CCA with respect to CS by reducing the time until there are enough faults that CCA outperforms CS. We also do not consider faults in the cache interface logic, which CCA could handle, and thus we slightly further bias our results against CCA.

4.2.2. *Fault-Free Single Core Performance*

A fault-free NC or CC pays a modest performance penalty due to the multiplexors (muxes) that determine from where each stage chooses its inputs. These muxes, which affect every pipeline stage, require a somewhat longer clock period to accommodate their latency. Also, CCA's additional area introduces some extra wiring delays, but the CAD tools revealed that this effect on the clock frequency was less than 0.3%.

The mux delays are identical for NCs and CCs, and they are not a function of the number of cores or number of CCs. In CCA3(2/1), each NC is choosing from among two inputs (itself or the CC). The CC is choosing from among three inputs (itself and both NCs), and thus has a 3-to-1 mux. However, at least one of those inputs is not changing, so the critical path of this 3-to-1 mux is the same as that of a 2-to-1 mux. In the other CCA chips, the NC and CC muxes are either 2-to-1 or 3-to-1, but we can leverage the same observation about non-changing inputs. Thus, in all CCA chips, each NC and each CC has a clock period penalty that is equal to the latency of one 2-to-1 mux. This clock period penalty is 4.5% in 90nm technology.

4.2.3. *Single NC Performance When Using CC*

An NC's use of cannibalized stages introduces some performance degradation. In Figure 6, we plot the performance of an NC in several situations: fault-free, using any immediate neighbor CC's stage and extending the clock period, and using a CC's stage and adding pipeline stages (i.e., for CCA4-pipe). Results are normalized to the performance (instructions per *second*) of a single baseline core that has none of CCA's added hardware. We compute wire delays based on prior work by Ho et al. [8], and we assume that the wires between NCs and CCs are routed using middle and upper metal layers. We used a modified version of the OpenRISC simulator to evaluate the IPC overhead for CCA4-pipe as a function of the cannibalized stage. We use the Media-

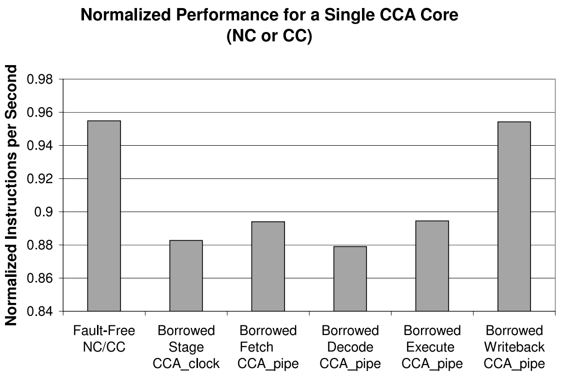


Figure 6. Performance of NC. With and without use of immediate neighbor CC (normalized to baseline core)

Bench benchmark suite [13] for these experiments, and the results are averaged over all of the benchmarks.

The results show that, when an NC borrows a CC's stage, the NC's slowdown is between 5% and 12%. Most slowdowns are in the 10-12% range, except when we add pipeline stages to borrow a Writeback stage; extending the Writeback stage incurs only a miniscule IPC penalty, because exceptions are rare.

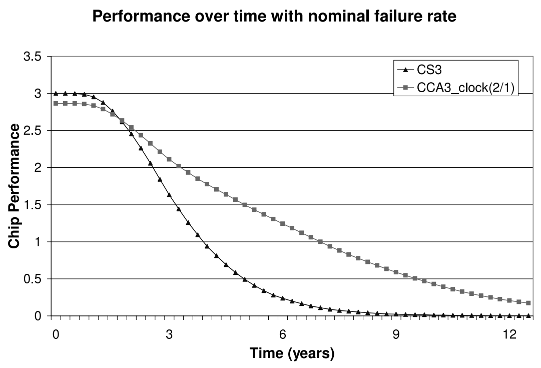
The performance when slowing the clock to accommodate a borrowed stage (the second bar from the left in Figure 6) is a function of the technology node. In Figure 6, we assumed 90nm technology. For larger/smaller CMOS technologies, the wire delays are smaller/greater [8]. Even at 45nm, the wire delays remain under 15% and 19% for immediate and non-immediate neighbors, respectively. Even a worst-case 19% clock degradation for a core is still preferable to disabling the core.

4.2.4. *Lifetime Multicore Performance*

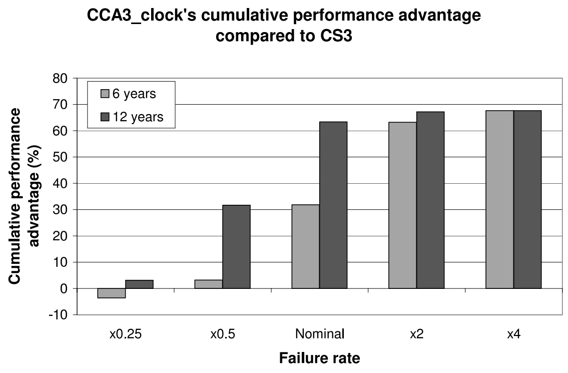
To determine aggregate multicore performance in the presence of faults, we developed Petri Net models of the CS and CCA chips. The Petri Net computes the expected performance of a chip as a function of time. We model each chip at the same 12-structure granularity as Blome et al. [2]. To evaluate a given chip, the Petri Net uses one million Monte Carlo simulations[1] in which we inject hard faults in each of the processor structures (including CCA logic and latches), using the distributions specified in Section 4.2.1. Once a fault occurs in a structure, the corresponding stage is considered unusable. For example, a fault in the ALU triggers the failure of the Execute stage. We do not consider the time needed to detect failures and reconfigure the chip.

We first evaluate chips with an equal number of cores and then evaluate equal-area chips. We report expected performance in units of "fault-free baseline processors." A CS3 chip with no faults has an expected performance of 3. CCA3(2/1) with no faults has an expected performance of 2.85, due to CCA3(2/1)'s clock penalty for mux delays. For

1. The results always converge well before the millionth simulation.

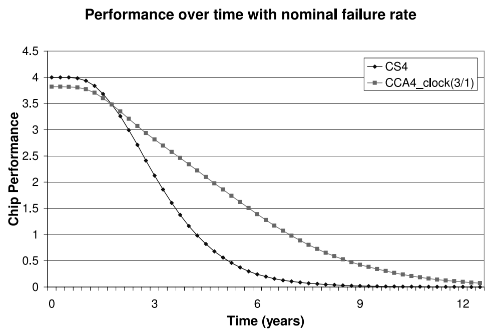


(a) Performance (nominal fault rates)

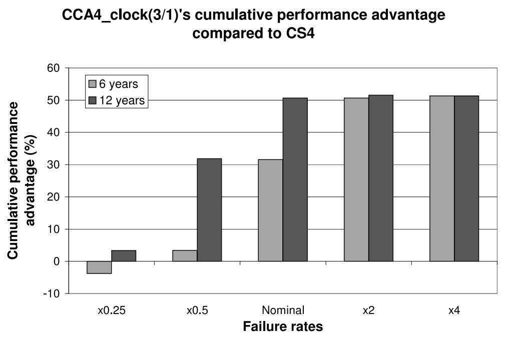


(b) Cumulative performance

Figure 7. Lifetime Performance of 3-core Chips



(a) CCA4-clock(3/1) Performance (nominal fault rates)



(b) CCA4-clock(3/1) Cumulative performance

Figure 8. Lifetime Performance of 4-core Chips

brevity, we refer to "expected performance" as simply "performance."

3-core Chips. Figure 7 plots performance over the lifetime of the chips. Figure 7a shows the performance of 3-core chips, assuming the nominal fault rates. The difference between the curves at time zero reflects CCA's fault-free performance overhead. We observe that the crossover point—the time at which the performances of CS3 and CCA3(2/1) are identical—is at a little under 2 years. After this early crossover point, CCA3(2/1)'s performance degradation is far less steep than CS3's. For example, after 6 years, CCA3(2/1) outperforms CS3 by one fault-free baseline core.

To better illustrate the importance of the gap between the curves in Figure 7a, Figure 7b shows the cumulative performance, for a variety of fault rates. The two bars for each fault rate represent the cumulative performance after 6 and 12 years, respectively. The cumulative performance is the integral (area under the curve) of the performance in Figure 7a. For nominal fault rates or greater, CCA3(2/1) provides substantially greater cumulative lifetime performance. After only 6 years at the nominal fault rate, CCA3(2/1) has a 30% advantage, and this advantage grows to over 60% by 12 years. Even at only half of the nominal fault rate, CCA3(2/1) has achieved a 30% improvement at 12 years. For very low fault rates, CCA3(2/1) has slightly less cumulative performance after 6 years and slightly more cumulative performance after 12 years, but neither difference is large.

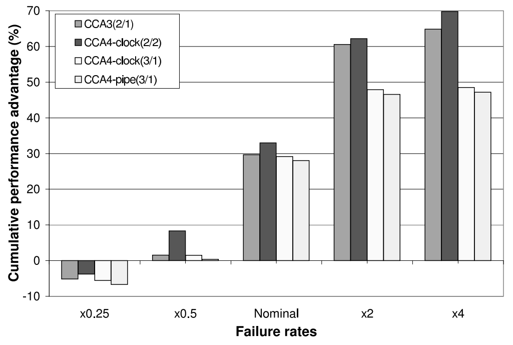
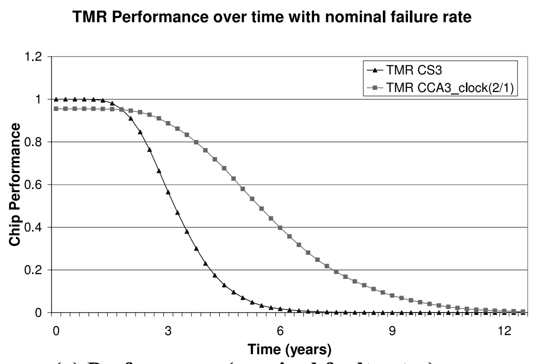


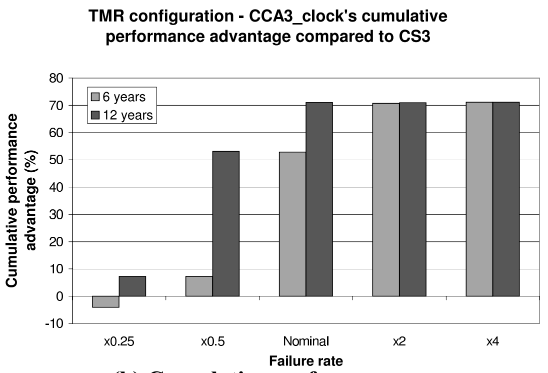
Figure 9. Equal-Area Lifetime Performance (6-year cumulative results)

4-core Chips. We present the results for 4-core chips in Figure 8. For CCA4-clock(3/1), Figure 8a shows that the crossover point is at approximately 2 years. Figure 8b shows that CCA4-clock(3/1) achieves a greater than 50% improvement in cumulative lifetime performance for the nominal and twice-nominal fault rates. Due to space constraints, we do not graph the results for CCA4-clock(2/2) and CCA4-pipe(3/1), but they exhibit very similar trends.

Equal-Area Comparisons. The 3-core and 4-core results presented thus far were not equal-area comparisons. CCA chips are slightly (less than 3.5%) larger than CS chips. To

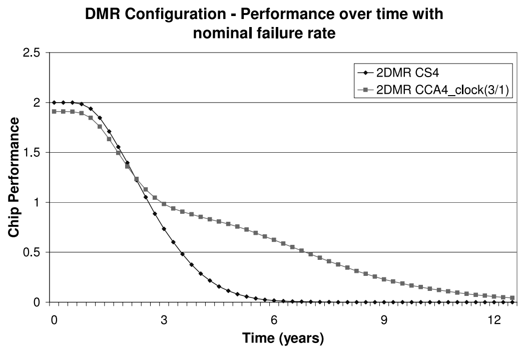


(a) Performance (nominal fault rates)

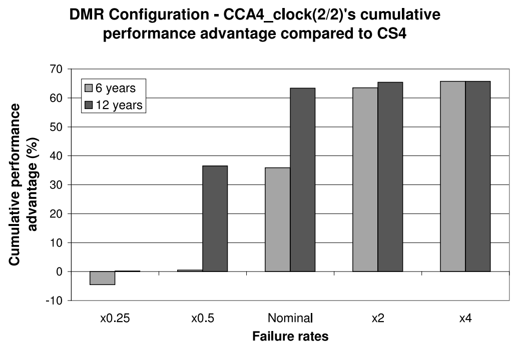


(b) Cumulative performance

Figure 10. Lifetime Performance of 3-core Chips Using TMR Cores



(a) CCA4-clock(2/2) Performance (nominal fault rates)



(b) CCA4-clock(2/2) Cumulative performance

Figure 11. Lifetime Performance of 4-core Chips Using Pairs of DMR Cores

provide another comparison point, we now compare chips of equal area. Note that the ratio of the chips' performances is independent of the chip size. Figure 9 plots the cumulative (over 6 years) performance advantages of the CCA chips. These results are quite similar to the earlier results, because CCA's area overheads are fairly small.

4.3. Performance of Chips Using TMR/DMR

If multiple cores are used to provide error detection with DMR or error correction with TMR, then CCA is beneficial as it allows for more cores to be available. We consider the performance of a chip to be the performance of the slowest core in a DMR or TMR configuration. If fewer than 2 cores are available, the chip has zero performance; we assume the user is unwilling to run without at least DMR to detect errors.

TMR. We plot the performance of 3-core chips that are being used in a TMR configuration in Figure 10. The crossover point is at about 2 years, similar to the comparison between CCA3 and CS3 when we were not considering TMR. However, the difference in cumulative performance is even greater. CCA3 provides more than 50% more cumulative performance for nominal and higher fault rates, even after only 6 years. At just half of the nominal fault rate, which is an optimistic assumption, CCA3 still has a 45% edge. The intuition for CCA's large advantage is that it greatly prolongs the chip's ability to operate in DMR mode. This analysis also applies to chips with more cores but where the cores are grouped into TMR clusters.

DMR. We consider the performance of 4-core chips that are comprised of two DMR pairs of cores (i.e., 4 cores total). The first fault in any core leads to the loss of one core and thus one DMR pair, for both CS4 and CCA4. Additional faults, however, are often tolerable with CCA4. Figure 11 shows the results for CCA4-clock(2/2), which is the best CCA4 design for this situation. Between approximately 2 and 2.5 years, CS4 and CCA4-clock(2/2) have similar performances. After that, though, CCA4-clock(2/2) significantly outperforms CS4. The cumulative results show that, for nominal and greater fault rates, CCA4-clock(2/2) provides lifetime advantages greater than 35% over 6 years and greater than 63% over 12 years.

5. RELATED WORK

We compare CCA to prior work in self-repair, pooling of core resources, and lifetime reliability.

5.1. Multicore-Specific Self-Repair

Multicore processors are inherently redundant, in that they contain multiple cores. Aggarwal et al. [1] proposed a reconfigurable approach to using multiple cores to provide redundant execution. When three cores are used to provide TMR, a hard fault in any given core will be masked. This use of redundant cores is related to the traditional fault tolerance schemes of multi-chip multiprocessors, such as IBM mainframes [19]. CCA is complementary to this work, in that CCA enables a larger fraction of on-chip cores to be available for TMR or DMR use. Concurrently with our work, Gupta et al. [6] have developed a single core in which the pipeline stages are connected by routers; this design could be used in a multicore chip to enable sharing of resources across cores. Such a multicore chip would enable greater flexibility in sharing than CCA, but it would incur a greater performance overhead for this flexibility.

5.2. Self-Repair for Superscalar Cores

Numerous researchers have observed that a superscalar core contains a significant amount of redundancy. Bower et al. [3] diagnose where a hard fault is—at the granularity of an ALU, reservation station, ROB entry, etc.—and deconfigure it. Shivakumar et al. [17] and Srinivasan et al. [22] similarly deconfigure components that are diagnosed by some other mechanism (e.g., post-fabrication testing). Rescue [15] deconfigures an entire "way" of a superscalar core if post-fabrication testing uncovers a fault in it. CCA differs from all of this work by targeting simple cores with little intra-core redundancy.

5.3. Pooling of Core Resources

There have been proposals to group cores together during phases of high ILP. Both Voltron [25] and Core Fusion [9] allow cores to be dynamically fused and un-fused to accommodate the software. These schemes both add a substantial amount of hardware, to allow tight coupling of cores, in pursuit of performance and power-efficiency. CCA differs from this work by being less invasive. CCA's goals are also different in that CCA seeks to improve lifetime performance.

5.4. Lifetime Reliability

Srinivasan et al. [20, 22] have explored ways to improve the lifetime reliability of a single superscalar core. These techniques include adding spare components, exploiting existing redundancy in a superscalar core, and adjusting voltage and frequency to avoid wearing out components too quickly. CCA is complementary to this work.

6. CONCLUSIONS

For multiprocessors with simple cores, there is an opportunity to improve lifetime performance by enabling sharing of resources in the presence of hard faults. The Core Cannibalization Architecture represents a class of designs that can retain performance and availability despite faults. Despite incurring slight performance overhead in fault-free scenarios, CCA's advantages over the course of time outweigh this initial disadvantage. From among the CCA designs we presented in this paper, we believe that CCA-clock designs are preferable to CCA-pipe designs. Even in those situations when CCA-pipe designs might eke out a bit more performance, it is not clear that their added complexity is worth this slight performance benefit. However, for future CMOS technologies, other core models, or cores with faster clocks, CCA-pipe may be worth its complexity.

Based on our results, we expect CCA (or designs like it) to flourish in two domains in particular. First, for many embedded applications, the key metric is availability at a reasonable performance, moreso than raw performance. Many embedded chips must stay available for long periods of time—longer than the average lifetime of a desktop, for example—and CCA improves this availability. Second, CCA's large benefits for chips that use cores in TMR and DMR configurations, suggest that CCA is a natural fit for chips using redundant cores to provide reliability.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant CCF-0444516, the National Aeronautics and Space Administration under grant NNG04GQ06G, Toyota InfoTechnology Center, and an equipment donation from Intel Corporation. We thank Alvy Lebeck, Anita Lungu, and Albert Meixner for their feedback on this work.

REFERENCES

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 470–481, June 2007.

[2] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating Online Wearout Detection. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[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 Annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–208, Nov. 2005.

[4] L. Carter, J. Feo, and A. Snavely. Performance and Programming Experience on the Tera MTA. In *Proceedings of the SIAM Conference on Parallel Processing*, Mar. 1999.

[5] Cisco Systems. Cisco Carrier Router System. http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdcco% nt_0900aecd800f8118.pdf, Oct. 2006.

[6] S. Gupta, S. Feng, J. Blome, and S. Mahlke. StageNetSlice: A Reconfigurable Microarchitecture Building Block for Resilient CMP Systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2008.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[8] R. Ho, K. W. Mai, and M. A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, Apr.

2001.

[9] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, June 2007.

[10] A. Iyer and D. Marculescu. Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 379–386, 2002.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[12] D. Lampret. OpenRISC 1200 IP Core Specification, Rev. 0.7. http://www.opencores.org, Sept. 2001.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[14] T. Nakura, K. Nose, and M. Mizuno. Fine-Grain Redundant Logic Using Defect-Prediction Flip-Flops. In *Proceedings of IEEE International Solid-State Circuits Conference*, 2007.

[15] 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*, pages 160–171, June 2005.

[16] M. Shah et al. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC. In *Proceedings of the IEEE Asian Solid-State Circuits Conference*, pages 22–25, Nov. 2007.

[17] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proceedings of the 21st International Conference on Computer Design*, Oct. 2003.

[18] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[19] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.

[20] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[21] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004.

[22] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[23] Y. Sugure et al. Low-Latency Superscalar and Small-Code-Size Microcontroller Core for Automotive, Industrial, and PC-Peripheral Applications. *IEICE Transactions on Electronics*, E89-C(6), June 2006.

[24] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing Co., 1982.

[25] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 25–36, Feb. 2007.