Putting Faulty Cores to Work

Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{ansary, shoe, shangupt, mahlke}@umich.edu

Abstract—Since the non-cache parts of a core are less regular, compared to on-chip caches, tolerating manufacturing defects in the processing core is a more challenging problem. Due to the lack of effective solutions, disabling non-functional cores is a common practice in industry, which results in a significant reduction in system throughput. Although a faulty core cannot be trusted to correctly execute programs, we observe that for most defects, when starting from a valid architectural state, execution traces on a defective core coarsely resemble those of fault-free executions. In light of this insight, we propose a robust and heterogeneous core coupling execution scheme, Necromancer, that exploits a functionally dead core to improve system throughput by supplying hints regarding high-level program behavior. We partition the cores in a CMP system into multiple groups in which each group shares a lightweight core that can be substantially accelerated using these execution hints from a faulty core.

Index Terms—Manufacturing defects, Wearout, Chip multiprocessors, Heterogeneous core coupling, System throughput

1 Introduction

Shrinking process technologies and rising power densities over the last decade have led to a host of reliability challenges such as defects, wear-out, and parametric variations [5]. One of these challenges for the semiconductor industry is manufacturing defects, which have a direct impact on yield. From each process generation to the next, microprocessors become more susceptible to manufacturing defects due to the higher sensitivity of materials, random particles attaching to the wafer surface, and sub-wavelength lithography issues. Thus, in order to maintain an acceptable level of manufacturing yield, a substantial investment is required. Traditionally, hardware reliability was only a concern for high-end systems (e.g., HP Tandem Nonstop) for which applying high-cost redundancy solutions such as triple modular redundancy (*TMR*) was acceptable. However, hardware reliability is emerging as a major issue for mainstream computing, where the usage of high-cost reliability solutions is not acceptable.

A large fraction of die area is devoted to caches, which can be protected using techniques like column redundancy. With appropriate protection mechanisms in place for caches, the processing cores become the major source of defect vulnerability on the die. Consequently, we try to tackle hard-faults in the non-cache portions of the processing core. Due to the inherent irregularity of the non-cache parts of the core, it is well-known that handling defects in these parts is challenging [10]. A common solution is to disable the faulty core [1]. However, industry is currently dominated by Chip Multi-Processor (*CMP*) systems with only a modest number of high-performance cores (e.g., Intel Core 2). Therefore, losing a core significantly reduces system throughput and sale price. At the other extreme of the solution spectrum lies fine-grained micro-architectural redundancy [12]. Unfortunately, since the majority of the core logic is non-redundant, the fault coverage from these approaches is very limited (e.g., less than 10% for an Intel processor [10]). StageNet [8] suggests breaking each core into pipeline stages and allowing one core to borrow stages from other cores through interconnection networks. Introduction of these interconnection networks in the processor pipeline presents performance, power consumption, and design complexity challenges. DIVA [2] was proposed for dynamic verification of complex microprocessors. It employs a checker pipeline that re-runs the same instruction stream to ensure correct program execution. Given the fact that DIVA was not

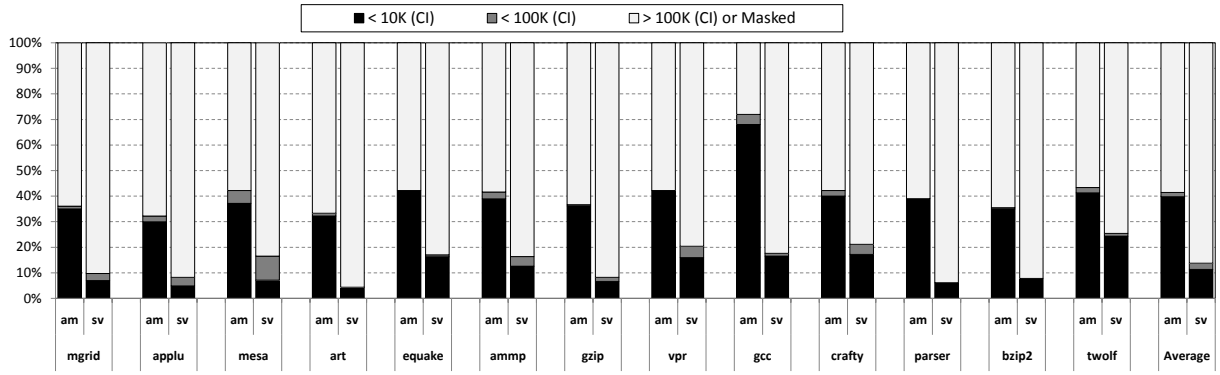

Fig. 1: Here, the first set of bars (*am*) shows the distribution of injected hard-faults that manifest as architectural state mismatches across different latencies, where latency is measured as the number of committed instructions (*CI*). The second set of bars (*sv*) shows the number of CIs before an injected fault results in a violation of a 90% similarity index.

intended to tolerate defects, as shown in [2], a hard-fault can result in about 10X slow-down.

As we will see, for most defect instances, the execution flow of the program on a faulty core *coarsely resembles* the fault-free program execution when starting from the same architectural state. Given this observation, we propose Necromancer (*NM*) to enhance overall system throughput and mitigate the performance loss caused by defects in the non-cache parts of the core. To accomplish this, we first relax the correct execution constraint on a faulty core (the *undead core*). Next, we leverage high-level execution information (*hints*) from the undead core to accelerate the execution of an *animator core*. The animator core is an additional core, introduced by NM, that is an older generation of the baseline cores. In the animator core, these hints are only treated as performance enhancers and do not influence execution correctness.

2 Motivation

In this section, we first demonstrate that an aggressive out-of-order (*OoO*) core with a hard-fault in the non-cache area cannot be trusted to operate correctly. Further, we test our hypothesis of whether it can still provide useful execution hints for another core.

2.1 Effect of Hard-Faults on Program Execution

In order to illustrate the negative impact of hard-faults on program execution, we identify the average number of instructions that can be committed before observing an architectural state mismatch. This result, for 5000 area-weighted hard-fault injections, is depicted as the first set of bars (*am*) in Figure 1. For these experiments, we have a golden execution which compares its architectural state with the faulty execution every cycle and as soon as a mismatch is detected, it terminates the simulation and reports the number of committed instructions up to that point. For instance, looking at equake, 42% of the injected hard-faults cause an architectural state mismatch in less than 10K committed instructions. As this figure shows, more than 40% of the injected hard-faults can cause an immediate architectural state mismatch. Thus, a faulty core cannot be trusted to provide correct functionality even for short periods of program execution.

2.2 Relaxing Correctness Constraints

Here, we try to determine the quality of program execution on a faulty core when relaxing the absolute correctness constraints. The second set of bars (*sv*) in Figure 1 depicts how many instructions can be committed in a faulty core before it gets considerably off the correct execution path. We define a similarity index (*SI*) that measures the similarity between the PC of committed instructions in the faulty core and a golden execution of the same program. This SI is calculated every 1K instructions and whenever it drops beneath a pre-specified threshold, we stop the simulation and record the number of committed instructions. Here, we

use a SI of 90% which means that during each 1K instruction window, 90% of PCs must hit exactly the same instruction cache line in both the golden execution and program execution on the faulty core. As can be seen, even for this high threshold value, in more than 85% of cases, the faulty core can successfully commit at least 100K instructions before its execution differs by more than 10%.

Since the execution behavior of a faulty core coarsely matches the golden program execution for long time periods, we can extract useful information from the execution of the program on the faulty core and send this information (hints) to the other core (the animator core), running the same program. This symbiotic relationship between the two cores enables the animator core to achieve a significantly higher performance. We allow the *undead core* to run without requiring absolutely correct functionality. Later, we will evaluate the performance boost that can be achieved by the NM system.

3 Challenges in Coupling with a Faulty Core

Given a CMP system, two cores can be coupled together to achieve higher single-thread performance. Since the overall performance of a coupled core system is bounded by the slower core, these two cores were traditionally identical. However, in order to accelerate program execution, one of these cores must progress through the program stream faster. In order to do so, three methods have been proposed. First, in Paceline [7], the core that runs ahead (*leader*) operates at a higher frequency than the core that receives execution hints (*checker*). When an architectural state mismatch happens, the frequency of the leader is adjusted. Alternatively, Slipstream processors [11] need two different versions of the same program. The leader core runs a shorter version based on the removal of ineffectual instructions while the checker core runs the unmodified program. Lastly, Flea-Flicker two pass pipelining [4] allows the leader core to return an invalid value on long-latency operations and proceed. In most of these schemes, the checker core takes advantage of program execution on the leader core by receiving preprocessed instruction streams, resolved branches, and L2 cache prefetches. Although a simple extension of these ideas seems plausible, due to the presence of defects, NM encounters two major difficulties.

Global Divergences: Hints become ineffective when the undead core gets completely off the correct execution path. To bring the undead core back to a valid execution point, the architectural state of the animator core can be copied over to the undead core. Although exact state matching, by checkpointing the register file, has been used [7], it is not applicable for animating a faulty core since architectural state mismatches occur so frequently. Therefore, we need coarse-grained online monitoring of the effectiveness of hints over a large time period to decide whether the undead core should be resynchronized with the animator core. Moreover, resynchronizations should be cheap and relatively infrequent to avoid a noticeable impact on the overall performance of the animator core.

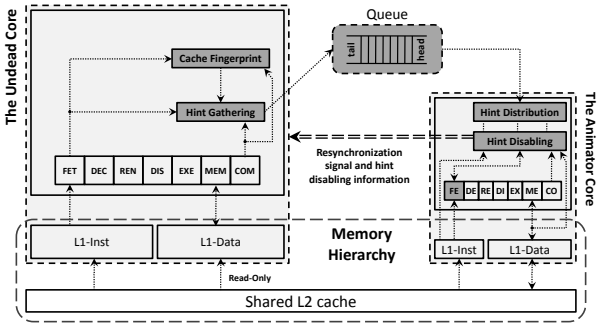

Fig. 2: The high-level architecture of NM with the modules that are added or modified highlighted.

Fine-Grained Variations: The *undead core* might execute/commit more or less number of instructions, causing variations in the similarity of program executions between the two cores. For instance, the undead core can take the wrong direction on an IF statement. Although it quickly returns to the correct execution path afterwards, a perfect data or instruction stream for the animator core is unattainable. This necessitates employing generic hints that are more resilient to these local abnormalities. Moreover, a mechanism is required to help the animator core identify the proper time for getting the hints from the dead core. Given the variation in the usefulness of the hints, in order to enhance the efficiency of the animator core, fine-grained hint disabling can also be leveraged.

4 NM Architecture

To mitigate system throughput loss due to defects, NM employs a robust and flexible heterogeneous core coupling technique. Given a group of cores, we introduce an animator core, an older generation with the same

ISA, that is shared among these cores. In this section, we describe the architectural details for a coupled pair of faulty and animator cores. As we saw earlier, by relaxing the correctness constraints, the undead core can execute a moderate portion of the program before a resynchronization is required. By executing the program on the undead core, NM provides hints to accelerate the animator core. In other words, the undead core is used as an external run-ahead engine for the animator core. Figure 2 illustrates the high-level NM design. In our design, most communication is unidirectional from the undead core to the animator core with the exception of the resynchronization and hint disabling signals. Consequently, a single queue is used for sending the hints and cache fingerprints to the animator core. The hint gathering unit attaches a tag to each queue entry to indicate its type. When this queue gets full and the undead core wants to insert a new entry, it stalls. To preserve correct memory state, we do not allow the dirty lines of the undead core's data cache to be written back to the shared L2 cache. Furthermore, exception handling is also disabled within the undead core since the animator core maintains the precise state.

In NM, we do not rely on overclocking the undead core or having multiple versions of the same program. Furthermore, NM is a hardware-based approach that is transparent to the workload and operating system. It also does not require register file checkpointing for performing exact state matching between two cores. Instead, we employ a fuzzy hint disabling approach based on the continuous monitoring of hint effectiveness, initiating resynchronizations when appropriate. In order to make the hints more robust against microarchitectural differences between the two cores and also variations in the number/order of executed instructions, we leverage the number of committed instructions for hint synchronization and attach this number to every queue entry as an *age tag*. Moreover, we introduce the concept of a *release window* to make the hints more robust in the presence of the aforementioned variations. The release window helps the animator core determine the right time to utilize a hint. For instance, assuming the data cache release window is 20, and 1030 instructions have already been committed in the animator core, data cache hints with age tags ≤ 1050 can be pulled off the queue and applied.

4.1 Hint Gathering and Distribution

Our branch prediction and cache hints (except L2 prefetches) need to be sent through the queue to the animator core. The hint gathering unit in the undead core is responsible for gathering hints and cache fingerprints, attaching the age and type tags, and finally inserting them into the queue. The PC of committed instructions and addresses of committed loads and stores are considered as hints. For branch prediction hints, the hint gathering unit sends a hint through the queue every time the branch predictor (*BP*) of the faulty core gets updates. On the animator core side, the hint distribution unit receives these packets from the queue and compares their age tag with the local number of committed instructions plus the corresponding release window sizes. It treats the incoming cache hints as prefetching information to warm-up its local caches.

The default BP of the animator core is a simple bimodal predictor. We first add an extra bimodal predictor (*NM BP*) to keep track of incoming branch prediction hints. Furthermore, we employ a hierarchical tournament predictor to decide, for a given PC, whether the original or NM BP should take over. As mentioned earlier, we leverage release windows to apply the hints just before they are needed. However, due to the variations in the number of executed instructions on the undead core, even the release window cannot guarantee perfect timing of the hints. In such a scenario, for a subset of instructions, the tournament predictor can give priority to the original BP of the animator core to avoid any performance penalty. Figure 3 shows a simple example in which the NM BP can only achieve 33% branch prediction accuracy. This is mainly due to the existence of a tight inner loop with a low trip count for which switching to the original BP can enhance the branch prediction accuracy.

In order to reduce the queue size, communication traffic needs to be limited to only the most beneficial hints. Consequently, in the hint gathering unit, we use two content addressable memories (*CAM*s) with several entries to discard I-cache and D-cache hints that were recently sent. Eliminating redundant hints also minimizes resource contention on the animator core. Furthermore, to save on transmission bits, we only send the block related bits of the address for cache hints, ignore hints on speculative paths, and for branch prediction hints, only send lower-order bits of the PC that are used for updating the branch history table of the NM BP.

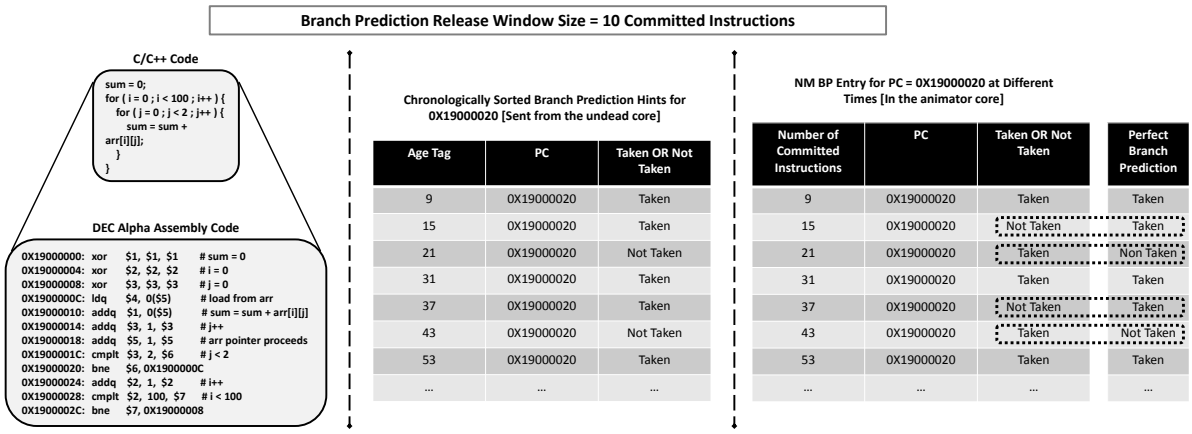

Fig. 3: A code example in which hints are received by the animator core at improper times, resulting in low branch prediction accuracy. Therefore, switching to the original BP of the animator core is beneficial. The code simply calculates the summation of a 2D-array elements. It should be noted that the branch prediction release window size is normally set so that the branch prediction accuracy for the entire execution gets maximized.

4.2 Why Disable Hints?

Hints can be disabled when they are no longer beneficial for the animator core. This might happen because the program execution on the undead core diverges from the correct execution path, performance of the animator core is already near its ideal case, or the undead core is not be able to get ahead of the animator core. In all these scenarios, hint disabling helps in four ways: (1) It avoids occupying resources of the animator core with ineffective hints. (2) The queue fills up less often, which means less stalls for the undead core. (3) Disabling hint gathering and distribution saves power. (4) It serves as an indicator of when the undead core has strayed far from the correct execution path and resynchronization is required.

4.3 Hint Disabling Mechanisms

The hint disabling unit is responsible for realizing when each type of hint should get disabled. In order to disable cache hints, the cache fingerprint unit generates high-level cache access information based on the committed instructions in the last time interval (e.g., last 1K committed instructions). These fingerprints are sent through the queue and compared with the animator core's cache access pattern. Based on a pre-specified threshold value for the similarity between access patterns, the animator core decides whether the cache hint should be disabled. In addition, when a hint gets disabled, the hint remains disabled throughout a significant time period, the back-off period.

Apart from prioritizing the original BP of the animator core for a subset of PCs, the NM BP can also be employed for global disabling of the branch prediction hints. For disabling branch prediction hints, we use a score-based scheme with a single counter. For every branch that the original and NM BPs agree upon no action should be taken. Nonetheless, for the branches that the NM BP correctly predicts and the original BP does not, the score counter is incremented by one. Similarly, for the ones that the NM BP mispredicts but the original BP correctly predicts, the score counter is decremented. Finally, at the end of each disabling time interval, if the counter is below a certain threshold, the branch prediction hints will be disabled for the back-off period.

4.4 Resynchronization

Since the undead core can stray from the correct execution path and no longer provide useful hints, a mechanism is required to restore it back to a valid state. To accomplish this, we occasionally resynchronize the two cores, during which the animator core's PC and architectural register values are copied to the undead core. According to [10], for a modern processor, this process takes on the order of 100 cycles. Moreover, all instructions in the undead core's pipeline are squashed, the rename table is reset, and the D-cache content is also

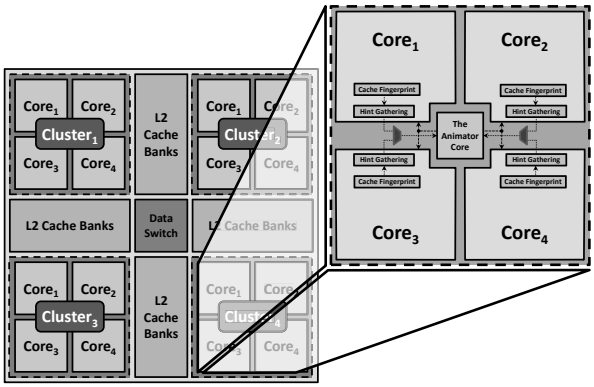

Fig. 4: The high-level NM design for a large CMP system with 16 cores, modeled after the Sun Rock processor.

invalidated. We take advantage of the hint disabling information to identify when resynchronization is needed. For instance, a potential resynchronization policy is to resynchronize when at least two of the hints get disabled.

4.5 NM Design for CMP Systems

Here, we describe how NM can be applied to CMP systems with more cores. Figure 4 illustrates the NM design for a 16-core system with 4 clusters modeled after the Sun Rock processor. Each cluster contains 4 cores which share a single animator core, shown in the call-out. In order to maintain scalability of the NM design, we employ the aforementioned 4-core cluster design as the basic building block. Since many dies are fault-free, in order to avoid disabling the animator cores, these cores can be leveraged for accelerating the operation of live cores by exploiting speculative method-level parallelism. However, evaluation of the latter is beyond the scope of this work.

For a heterogeneous CMP system, sharing an animator core between multiple cores might not be possible since cores have different computational capabilities. A potential solution is to partition the original set of cores into groups such that in each group, we have several large cores and a small core. In each group, the smaller core should have the capability of operating as a conventional core or as an animator core when there is a defect in one of the larger cores within its group. These dual purpose cores are a suitable fit for many heterogeneous CMP systems that are designed with many simple cores such as the IBM Cell processor.

5 Evaluation Methodology

We heavily modified SimAlpha [3] to model our coupled core execution. Inter-process communication is used to model the information flow between our 6-issue OoO Alpha 21264 and 2-issue OoO core with the same resources as Alpha 21064 while simulating SPEC-CPU-2K. To study the effect of manufacturing defects on the NM system, we developed an area-weighted, Monte Carlo fault injection engine. During each iteration of the Monte Carlo simulation, a microarchitectural structure is selected and a single, random stuck-at fault is injected into the timing simulator. We inject these hard-faults in 15 different microarchitectural structures shown in Figure 5. Dynamic, and static, power consumption is evaluated using Wattch [6], HotLeakage [13] and CACTI [9]. The Synopsys standard industrial tool-chain, with a TSMC 90*nm* technology library, is used to evaluate the overheads of the remaining miscellaneous logic (e.g. shift registers).

6 Results

As described earlier, there are many parameters in the NM design such as cache hint CAM sizes, release window sizes, and hint disabling thresholds. To fix these architectural parameters, we performed an extensive design space exploration. Given these parameter values, on average, NM can achieve 39.5% speed-up over the baseline animator core.

Impact of hard-fault locations on performance: To highlight the impact of fault locations on the achievable speed-up by NM, Figure 5 depicts the performance breakdown for 15 different fault locations in the Alpha 21264 ipeline. Results in each column are normalized to the average speed-up that can be achieved by NM for that particular benchmark. This was done to eliminate the advantage/disadvantage that comes from the inherent benchmark suitability for core coupling. As can be seen, hard-faults in some locations are more harmful than others. These locations consist of the PC, integer ALU, and instruction fetch queue. Another interesting observation is that reaction to defects can significantly differ between benchmarks (e.g., parser). We conclude two main points from this plot. First, on average, there are only a few fault locations that can drastically impact

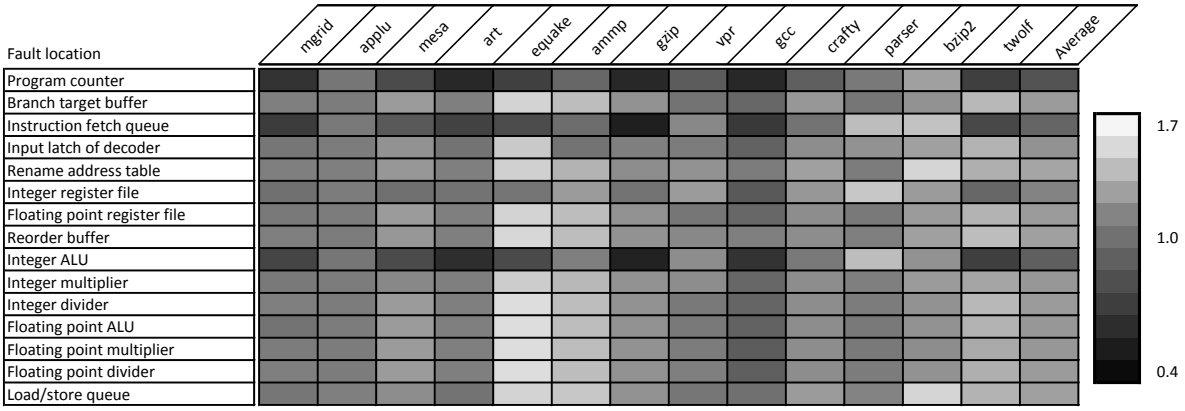

Fig. 5: Variation in the speed-up of the animator core for different hard-fault locations across SPEC-CPU-2K benchmarks. To isolate the impact of hard-fault locations, results in each column are normalized to the average speed-up that can be achieved by the NM coupled cores for that particular benchmark.

NM speed-up gains. Second, for a given fault location, different benchmarks show various degrees of susceptibility; thus, heterogeneity across the benchmarks running on a CMP system helps NM to achieve a higher speed-up by allowing more suitable workloads to be assigned to the coupled cores.

Speed-up and overheads: Table 1 demonstrates the amount of speed-up that can be achieved by the NM coupled cores. We achieve a higher overall speed-up as the number of cores increases. This is because NM achieves different speed-ups based on the defect type, location, and the workload running on the system. For a 16-core system, on average, the coupled cores can achieve the performance of a live core, essentially providing the appearance of a fully-functional 6-issue baseline core with a 2-issue animator core. Here, we assume full utilization, which means there is always one job per core. Hence, for larger CMPs, with more heterogeneity across the benchmarks running on the system, there is more opportunity for NM to exploit. This table also shows the area and power overheads for our scheme. As can be seen, the area overhead gradually shrinks as the number of cores grows since the cost of the animator core is amortized among more cores. Nevertheless, since we simply replicate the 4-core building block to construct CMPs with more than 4 cores, the area overhead remains the same. In terms of power overhead, as the speed-up results show, for CMPs with less than 8 cores, the undead core remains ahead of the animator core and must stall when the queue is full. During stalls, the undead core does not consume dynamic power.

Throughput enhancement: The main objective of

TABLE 1: Summary of performance benefit, area overhead, and power overhead of our scheme for CMP systems with different numbers of cores. Performance of NM coupled cores is normalized to the average performance of a baseline animator core and also to the average performance of a live core.

Target system	**Performance of NM norm. to a**		**Area overhead (percentage)**	**Power overhead (percentage)**
	baseline animator core	**live core**		
Single-core (2MB L2)	1.39	0.68	13.6%	15.6%
2-core CMP (2MB shared L2)	1.59	0.78	10.1%	11.8%
4-core CMP (4MB shared L2)	1.79	0.87	5.3%	8.5%
8-core CMP (8MB shared L2)	1.94	0.95	5.3%	5.1%
16-core CMP (16MB shared L2)	2.02	0.99	5.3%	2.7%

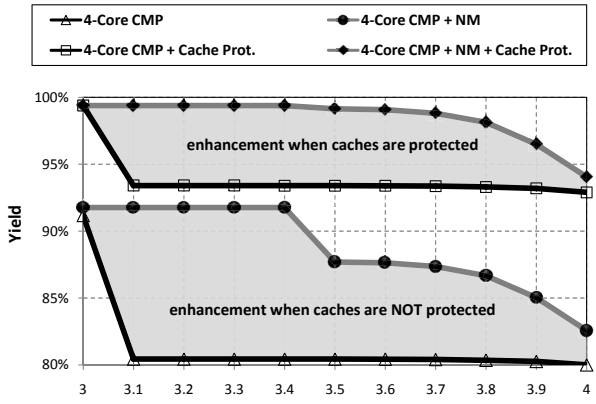

Fig. 6: Achievable yield for a 4-core CMP, given an expected level of system throughput. Here, we consider two baselines, a CMP system with and without proper protection for on-chip caches, showing the yield improvement when NM is applied.

NM is to improve the average system throughput of a population of manufactured chips. For this purpose, we model 1000 manufactured chips with randomly distributed defects based on our target defect rate. In the case of a defect in one of the original cores, we apply our scheme. On the other hand, if any of the animator cores, communication queues, or NM specific modules like the hint gathering unit are faulty, we simply disable the animator core and the rest of the system can continue their normal operation. Figure 6 depicts the throughput enhancement results (shaded regions) based on *throughput binning* for a 4-core CMP. Note that NM significantly enhances the overall system throughput. The horizontal axis shows the system throughput, normalized to the throughput of a single baseline core. In this plot, we illustrate the throughput binning results for throughput values between 3 and 4. Since we assume, on average, one defect per 5 chips, yield is always above 80%. However, there is a small chance that multiple defects hit the same chip, which precludes a yield of 100% at a throughput of 3, even when protecting the on-chip caches. As can be seen, cache protection is a necessity and fortunately, can be provided easily (e.g., column redundancy).

7 Summary

To maintain an acceptable level of yield in nanoscale technologies, manufacturing defects need to be addressed properly. Since non-cache parts of a core are less structured and homogeneous, tolerating defects in the general core area has remained a challenging problem. In this work, we presented Necromancer, an architectural scheme to enhance system throughput by exploiting faulty cores. Necromancer does not rely on correct program execution on a faulty core; instead, it only expects this undead core to generate effective execution hints to accelerate an animator core. In order to increase Necromancer's efficacy, we use microarchitectural techniques to provide intrinsically robust hints, effective hint disabling, and dynamic inter-core state resynchronization.

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 *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 470–481, 2007.

[2] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[4] R. D. Barnes, E. N. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, page 387, 2003.

[5] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[6] D. Brooks, V. Tiwari, and M. Martonosi. A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[7] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 213–224, 2007.

[8] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 141–151, 2008.

[9] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *IEEE Micro*, pages 3–14, 2007.

[10] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[11] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 269–280, 2000.

[12] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 520–531, June 2005.

[13] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical report, Univ. of Virginia Dept. of Computer Science, Jan. 2003.