Necromancer: Enhancing System Throughput by Animating Dead Cores

Amin Ansari Shuguang Feng Shantanu Gupta Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{ansary, shoe, shangupt, mahlke}@umich.edu

ABSTRACT

Aggressive technology scaling into the nanometer regime has led to a host of reliability challenges in the last several years. Unlike on-chip caches, which can be efficiently protected using conventional schemes, the general core area is less homogeneous and structured, making tolerating defects a much more challenging problem. Due to the lack of effective solutions, disabling non-functional cores is a common practice in industry to enhance manufacturing yield, which results in a significant reduction in system throughput. Although a faulty core cannot be trusted to correctly execute programs, we observe in this work that for most defects, when starting from a valid architectural state, execution traces on a defective core actually 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 conventional CMP system into multiple groups in which each group shares a lightweight core that can be substantially accelerated using these execution hints from a potentially dead core. To prevent this *undead* core from wandering too far from the correct path of execution, we dynamically resynchronize architectural state with the lightweight core. For a 4-core CMP system, on average, our approach enables the coupled core to achieve 87.6% of the performance of a fully functioning core. This defect tolerance and throughput enhancement comes at modest area and power overheads of 5.3% and 8.5%, respectively.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Design, Reliability, Performance

Keywords

Manufacturing defects, Heterogeneous core coupling, Execution abstraction

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

1. INTRODUCTION

The rapid growth of the silicon process over the last decade has substantially improved semiconductor integration levels. However, this aggressive technology scaling has lead to a host of reliability challenges such as manufacturing defects, wear-out, and parametric variations [10, 9]. These threats can affect correct program execution, one of the most significant aspects of any computer system [4]. Traditionally, hardware reliability was only a concern for high-end systems (e.g., HP Tandem Nonstop and IBM eServer zSeries) for which applying high-cost redundancy solutions such as triple modular redundancy (*TMR*) was acceptable. Nevertheless, hardware reliability has already become a major issue for mainstream computing, where the usage of high-cost reliability solutions is not acceptable [24].

One of the main 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 higher sensitivity of materials, random particles attaching to the wafer surface, and sub-wavelength lithography issues such as exposure tool optimization, cleaning technology, and resist process optimization [18]. Thus, in order to maintain an acceptable level of manufacturing yield, a substantial investment is required [32]. Traditionally, modern high-performance processors are declared as functional if all parts of the design are fault-free, or if they can operate correctly by tolerating failures. However, since manufacturing defects can cause a significant yield loss, semiconductor companies have recently started to manufacture parts that have been over-designed to hedge against defects. For instance, to improve yield, IBM did this with the Cell Broadband Engine that sometimes only had 7 out of the 8 processing elements activated [34].

Based on the latest ITRS report [19], for current and near future CMOS technology, one manufacturing defect per five $100mm^2$ dies can be expected. Fortunately, a large fraction of die area is devoted to memory structures, in particular caches, which can be protected using existing techniques such as row/column redundancy, 2D-ECC [21], ZerehCache [3], Bit-Fix [38], and sub-block disabling [1]. 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 parts of the processing core. Due to the inherent irregularity of the general core area, it is well-known that handling defects in the non-cache parts is challenging [27]. A common solution is core disabling [2]. However, the industry is currently dominated by Chip Multi-Processor (*CMP*) systems with only a modest number of high-performance cores (e.g., Intel Core 2), systems which cannot afford to lose a core due to manufacturing defects. The other extreme of the solution spectrum lies fine-grained

micro-architectural redundancy [32, 12, 35]. Here, broken micro-architectural structures, such as ALUs, are isolated or replaced to maintain core functionality. Unfortunately, since the majority of the core logic is non-redundant, the fault coverage from these approaches is very limited – less than 10% for an Intel processor [27].

In this work, we propose Necromancer (*NM*) to tackle manufacturing defects in current and near future technology nodes. NM enhances overall system throughput and mitigates 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* – since it cannot be trusted to faithfully execute programs. 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 CMP with less resources and the same instruction set architecture (*ISA*). The main rationale behind our approach is the fact that, for most defect instances, the execution flow of the program on the undead core *coarsely resembles* the fault-free program execution on the animator core – when starting from the same architectural state (i.e., program counter (*PC*), architectural registers, and memory). Moreover, in the animator core, these hints are only treated as performance enhancers and do not influence execution correctness. In NM, we rely on intrinsically robust hints and effective hint disabling to ensure the animator core is not mislead by unprofitable hints. Dynamic inter-core state resynchronization is also employed to update the undead core with valid architectural state whenever it strays too far from the correct execution path. To increase our design efficiency, we share each small animator core among multiple cores. Our scheme is unique in the sense that it keeps the undead core on a semi-correct execution path, ultimately enabling the animator core to achieve a performance close to the performance of a live (fully-functional) core. In addition, NM does not noticeably increase the design complexity of the baseline cores and can be easily applied to current and near future CMP systems to enhance overall system throughput.

2. UTILITY OF AN UNDEAD CORE

We motivate the NM design by demonstrating the high-level rationale behind it. To this end, we provide evidence that supports the following two statements: **(1)** Although an aggressive out-of-order (*OoO*) core with a hard-fault in the non-cache area cannot be trusted to perform its normal operation, it can still provide useful execution hints in most cases. **(2)** By exploiting hints from the undead core, the animator core can typically achieve a significantly higher performance.

2.1 Effect of Hard-Faults on Program Execution

Prior work has studied the effect of a single-event upset, or a transient fault, on program execution for high-performance microprocessors. Using fault-injection, it has been shown that transient faults are often masked, easier to categorize, and have a temporal effect on program behavior [37]. On the other hand, the effect of hard-faults on program execution is hard to study since each hard-fault can result in a complicated intertwined behavior. For example, a hard-fault can cause multiple data corruptions that finally mask each others effect. Moreover, hard-faults are persistent and their effect does not go away. As a result, hard-faults can dramatically corrupt program execution. In order to illustrate the negative impact of hard-faults on program execution, we study the average number of instructions that can be committed before observing an architectural state mismatch. This result, for 5000 area-weighted hard-fault injection experiments across SPEC-CPU-2K benchmarks, is depicted in Figure 1. Details of the Monte Carlo engine, statistical area-weighted fault injection infrastructure, target system, and benchmark suite can be found in Section 5.1. 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 stops the simulation and reports the number of committed instructions up to that point. For instance, looking at 188.amp, 26% of the injected hard-faults cause an architectural state mismatch to happen in less than 100 committed instructions. Since 176.gcc more uniformly stresses different core resources, it shows a higher vulnerability to hard-faults. As this figure shows, more than 40% of the injected hard-faults can cause an immediate – $<$ 10K – architectural state mismatch. Thus, a faulty core cannot be trusted to provide correct functionality even for short periods of program execution.

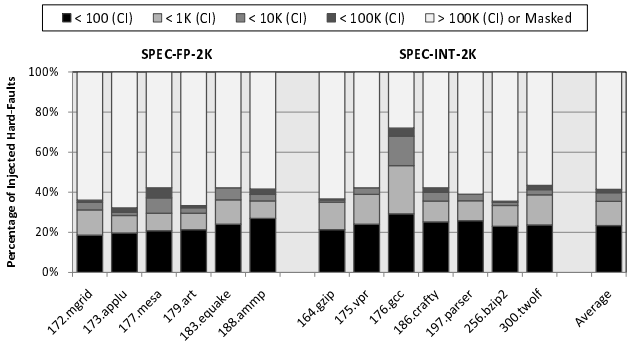


Figure 1: Distribution of injected hard-faults that manifest as architectural state mismatches across different latencies – in terms of the number of committed instructions (*CI*).

2.2 Relaxing Correctness Constraints

As just discussed, program execution on a dead core cannot be trusted. Here, we try to determine the quality of program execution on a dead core when relaxing the absolute correctness constraints. In other words, we are interested in knowing for what expected level of correctness, a dead core can practically execute large chunks of a program. Based on 5K injected hard-faults, Figure 2 depicts how many instructions can be committed in a dead core before it gets considerably off the correct execution path. In order to have a practical system, the dead core should be able to execute the program over reasonable time periods before its execution becomes ineffectual. Here, we define a similarity index (*SI*) that measures the similarity between the PC of committed instructions in the dead core and a golden execution of the same program. This SI is calculated every 1K instructions and whenever it becomes less than a pre-specified threshold, we stop the simulation and record the number of committed instructions. For instance, a similarity index of 30% for PC values means, that during each 1K instruction window, 30% of PCs hit exactly the same instruction cache line in both the golden execution and program execution on the dead core. Figure 2 shows the number of committed instructions for three different SI thresholds. For instance, considering SI threshold of 90%, on average only 12% of the hard-faults renders the program execution on a dead core ineffectual before at least 10K instructions get committed. Hence, even for an SI threshold of 90%, in more than 85% of cases, the dead core can successfully commit at least 100K instructions before its execution differs by more than 10%.

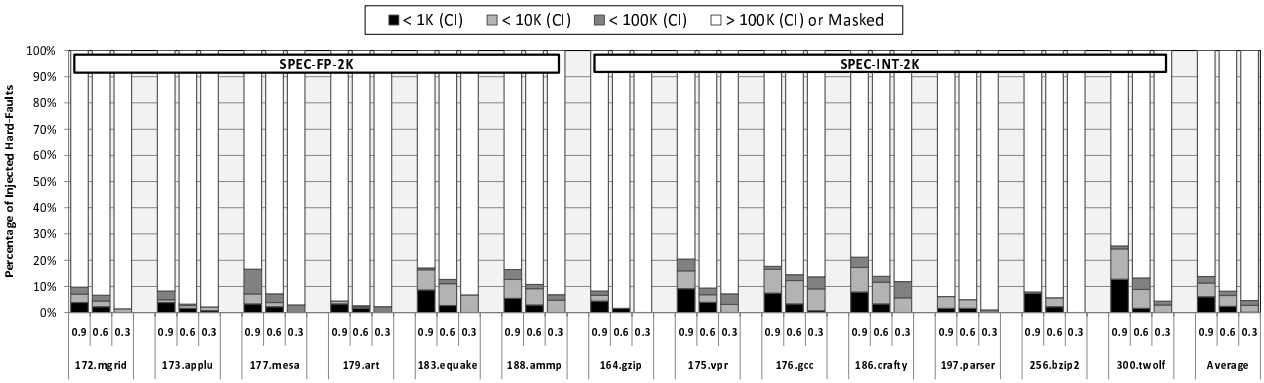


Figure 2: Number of instructions that are committed (*CI*) before an injected hard-fault results in a violation of a pre-specified similarity index threshold. For this purpose, 5K hard-faults were injected while considering three different similarity index thresholds (90%, 60%, and 30%).

2.3 Opportunities for Acceleration

Since the execution behavior of a dead core coarsely matches the intact program execution for long time periods, we can take advantage of the program execution on the dead core to accelerate the execution of the same program on another core. This can be done by extracting useful information from the execution of the program on the dead core and sending this information (hints) to the other core (the animator core), running the same program. We allow the *undead core* to run without requiring absolutely correct functionality. The undead core is only responsible to provide helpful hints for the animator core. This symbiotic relation between the two cores enables the animator core to achieve a significantly higher performance. When the hints lose their effectiveness, we resynchronize the architectural state of the two cores. Since an architectural state resynchronization, between two cores in a CMP system, takes about 100 cycles [27] and resynchronization in more than 85% of cases happens after at least 100K committed instructions, the overhead associated with resynchronization is small.

For the purpose of evaluation and since we want to have a single ISA system, based on the availability of the data on the power, area, and other characteristics of microprocessors, we use an EV6 (DEC Alpha 21264 [20]) for the baseline cores. On the other hand, for the animator core, we select a simpler core like the EV4 (DEC Alpha 21064) or EV5 (DEC Alpha 21164) to save on the overheads of adding this extra core to the CMP system. In order to evaluate the efficacy of the hints, in Figure 3, we show the performance boost for the aforementioned DEC Alpha cores using perfect hints (*PHs*) – perfect branch prediction and no L1 cache miss. Here, we have also considered the EV4 (OoO), an OoO version of the 2-issue EV4, as a potential option for our animator core. As can be seen, by employing perfect hints, the EV4 (OoO) can outperform the 6-issue OoO EV6 in most cases; thus, demonstrating the possibility of achieving a performance close to the performance of a live core through the NM system. Nevertheless, achieving this goal is quite challenging due to the presence of defects, different sources of imperfection in hints, and inter-core communication issues.

3. FROM TRADITIONAL COUPLING TO ANIMATION

In a CMP system, prior work has shown 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 to sustain an acceptable level of single-thread performance. However, in order to accelerate program execution, one of these coupled cores must progress through the program stream faster than the other. In order to do so, three methods have been proposed:

- In Paceline [16], the core that runs ahead (*leader*) and the core that receives execution hints (*checker*) from the leader core operate at different frequencies. Paceline cuts the frequency safety margin of the leader core and continuously compares the architectural state (excluding memories) of the two cores. When a mismatch happens, the frequency of the leader is adjusted, L1 state match is enforced, and finally the checkpoint interval is rolled back for re-execution.
- Slipstream processors [28] and Master/Slave speculative parallelization [41] need two different versions of the same program. In these schemes, the leader core runs a shorter version of the program based on the removal of ineffectual instructions while the checker core runs the unmodified program.
- Finally, Flea-Flicker two pass pipelining [6] and Dual-Core Execution [40] allow the leader core to return an invalid value on long-latency operations and proceed.

Although these schemes have widely varying implementation details, they share some common traits. In these schemes, the leader core tries to get ahead and sends hints that can accelerate checker core execution. These two cores are connected through one/several first-in first-out (*FIFO*) hardware queues to transfer hints and retired instructions along with their PCs. The checker core takes advantage of program execution on the leader core in 3 ways. First, the checker core receives pre-processed instruction and data streams. Second, during the program execution in the leader core, most branch mispredictions get resolved. Third, the program execution in the leader core automatically initiates L2 cache prefetches for the checker core.

A straight-forward extension of these ideas to animate a dead core seems plausible. However, NM encounters major difficulties when trying to fit the dead core into this execution model. Here, we briefly describe the two main challenges, leaving discussions of the proposed microarchitectural solutions for subsequent sections.

Fine-Grained Variations: One of the main sources of problems is the presence of defects in the dead core. Due to the presence of defects, 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, in many cases, the undead core can take the wrong direction on an IF statement and get back to the right execution path afterwards, thereby preventing

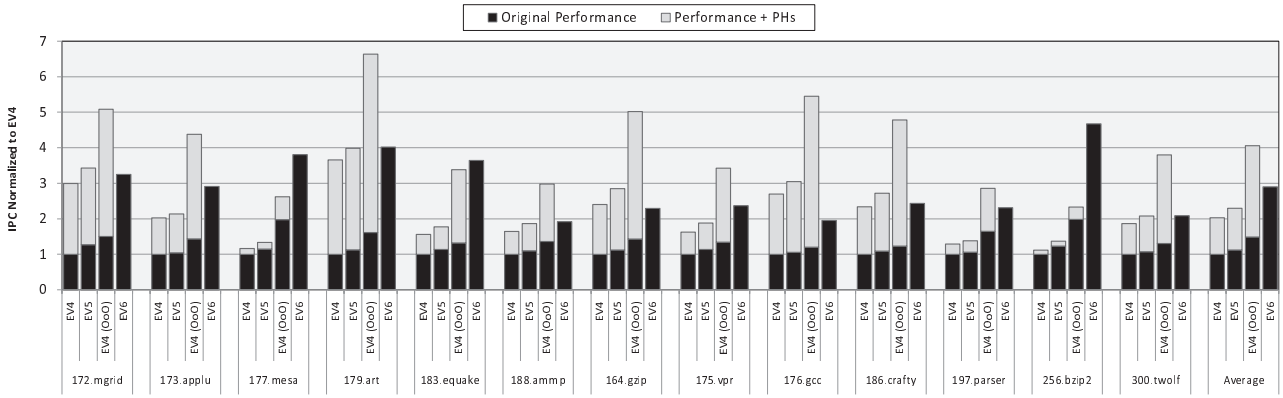


Figure 3: IPC of different DEC Alpha microprocessors, normalized to EV4's IPC. In most cases, by providing perfect hints for the simpler cores (EV4, EV5, and EV4 (OoO)), these cores can achieve a performance comparable to that achieved by a 6-issue OoO EV6.

a perfect data or instruction stream for the animator core. This necessitates employing generic hints that are more resilient to these local abnormalities. Moreover, the number of times that each PC is visited cannot be used to synchronize the two cores. A mechanism is required to help the animator core identify the proper time for pulling the hints off the communication queue. Given the variation in the usefulness of the hints, in order to enhance the efficiency of the animator core, fine-grained hint disabling can be leveraged. For instance, if the last K branch prediction hints for a particular PC were not useful, branch prediction for this particular PC can be handled by the animator core's branch predictor.

Global Divergences: When the undead core gets completely off the correct execution path, hints become useless, and it needs to be brought back to a valid execution point. For this purpose, 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 in prior work [16], it is not applicable for animating a dead core since architectural state mismatches occur so frequently. Therefore, we need coarse-grained online monitoring of the effectiveness of the 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. One possible approach for maintaining correct memory state, suggested by Paceline, is to re-fetch the cache-lines that are accessed during the last checkpointed interval into the L1 cache of the leader core [16]. However, since this might happen often for a dead core, we need a low-cost resynchronization approach that does not require substantial book keeping.

4. NM ARCHITECTURE

The main objective of NM is to mitigate system throughput loss due to manufacturing defects. For this purpose, it leverages a robust and flexible heterogeneous core coupling execution technique which will be discussed in the rest of this section. Given a group of cores, we introduce an animator core, an older generation with the same ISA, that is shared among these cores for defect tolerance purposes. In this section, we describe the architectural details for a coupled pair of dead and animator cores. The high-level NM design for a CMP system with more cores will be discussed in the next section. In Section 2, we showed that the faulty core – the undead core – cannot be trusted to run even a short part of the program. However, as we relaxed the exact architectural state match and looked at the global execution pattern, 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 without requiring multiple versions of the same program. In other words, the undead core is used as an external run-ahead engine for the animator core that has been added to the CMP system. We believe NM is a valuable solution for improving the system throughput of the current and near future mainstream CMP systems without notably influencing design complexity.

4.1 High-Level NM System Description

Figure 4 illustrates the high-level NM heterogeneous coupled core design. As discussed in Section 2, for the purpose of evaluation, we use 6-issue OoO EV6 for the baseline cores and a 2-issue OoO EV4 as our animator core. In our design, most communications are unidirectional from the undead core to the animator core with the exception of the resynchronization and hint disabling signals. Thus, a single queue is used for sending the hints and cache fingerprints to the animator core. The hint gathering unit attaches a 3-bit 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. As a result, a dirty data cache-line of the undead core is simply dropped whenever it requires replacement. Exception han-

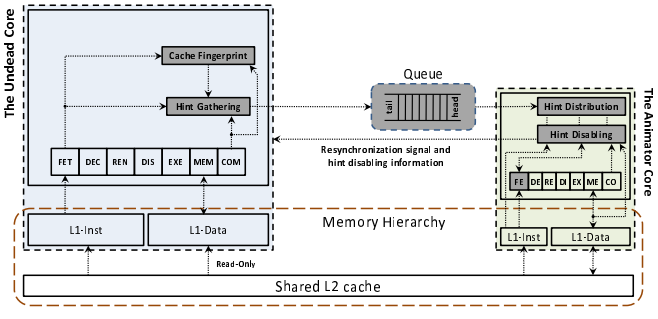
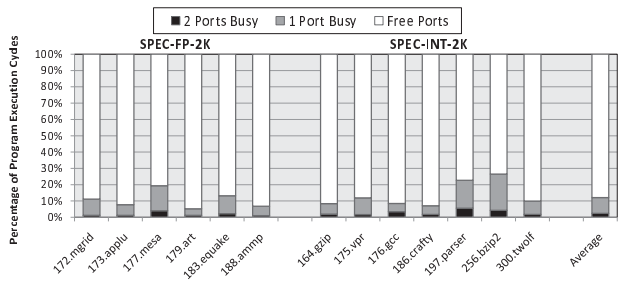
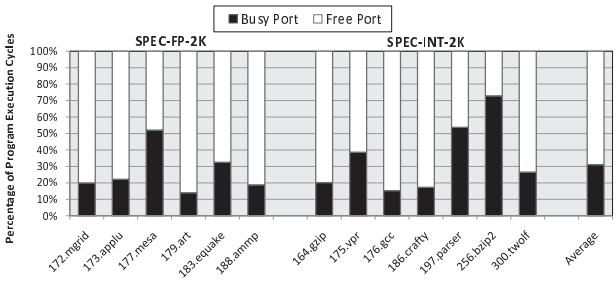


Figure 4: The high-level architecture of NM is shown in this figure and modules that are modified or added to the underlying cores are highlighted (not drawn to scale).



(a) Port activity for the animator core's L1-data cache



(b) Port activity for the animator core's L1-instruction cache

Figure 5: Port activity breakdown for local caches of the animator core. Here, we show the percentage of cycles that each cache port is either busy or free. For our animator core, the data cache has 2 ports while the instruction cache has a single port.

dling is also disabled at the undead core since the animator core maintains the precise state.

As discussed in Section 2, the animator core with perfect hints has the potential of surpassing the average performance of a live core. Nonetheless, the performance of the undead core can be a bottleneck for the NM system since: **a.** In many cases (Figure 3), performance of a baseline core is worse than the performance of the animator core with perfect hints. **b.** After each resynchronization, the undead core needs to warm-up the branch predictor and local caches. Therefore, we allow the undead core to proceed on the data cache L2 misses, without waiting for the several hundred cycles needed to receive data back from main memory. We simply return zero since L2 misses are not common and also value prediction would not be beneficial. This has a large impact on the performance of the undead core, potentially shortening the resynchronization period. Given the ability to eliminate stalls on L2 misses and also semi-perfect hints from the undead core, NM can *potentially* achieve even a higher performance than that of a live core. Nevertheless, providing even semi-perfect hints is challenging due to defects in the undead core, queue size, limited performance of the undead core, queue delay, and natural fluctuations in program behavior.

NM uses a heterogeneous core coupling program execution with a pruned core that has a significantly smaller area compared to a baseline core. In NM, we do not rely on overclocking the undead core or having multiple versions of the same program. Furthermore, it is a hardware-based approach that is transparent to the workload and operating system (*OS*). 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 the hints effectiveness, and initiating resynchronizations when appropriate. Hint disabling also helps to enhance performance and save on communication power for program phases in which the undead core cannot get ahead of the animator core. Apart from that, the undead core might occasionally get off the correct execution path (e.g., taking the wrong direction on an IF statement) and return to the correct path afterwards – Y-branches [36]. In order to make the hints more robust against microarchitectural differences between 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 *release window* concept to make the hints more robust in the presence of aforementioned variations. For a particular hint type, the release window helps the animator core to determine the right time to utilize a hint. For instance, assuming the data cache (*D-cache*) release window is 100, and 1000 instructions have already been committed in the animator core, D-cache hints with age tags ≤ 1100 can be pulled off the queue and applied.

4.2 Hint Gathering and Distribution

Program execution on the undead core automatically warms-up the shared L2 cache without requiring communication between two cores. However, other hints – i.e., L1 data cache, L1 instruction cache, and branch prediction hints – 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 3-bit type tags, and finally inserting them into the queue. On the other side, the hint distribution unit receives these packets and compares their age tag with the local number of committed instructions plus the corresponding release window sizes.

Every cycle, the hint gathering unit looks over the committed instructions for data and instruction cache (*I-cache*) hints. In fact, the PC of committed instructions and addresses of committed loads and stores are considered as I-cache and D-cache hints, respectively. On the animator core side, the hint distribution unit treats the incoming I-cache and D-cache hints as prefetching information to warm-up its local caches. For the animator core, Figure 5 depicts the utilization of two D-cache ports and a single I-cache port. Given the pipelined cache access for all high-performance processors, as can be seen for D-cache, both ports are busy for less than 5% of cycles. Therefore, we leverage the original cache ports for applying our D-cache hints. However, since hints can only *potentially* help the program execution, priority of the access should always be given to the normal operation of the animator core. On the other hand, the I-cache port is busy for more than 50% of cycles for 3 benchmarks and is free only if the instruction fetch queue (*IFQ*) is full. Moreover, since the I-cache operation is critical for having a sustainable performance, we add an extra port to this cache in the animator core.

In order to provide branch prediction hints, the hint gathering unit looks at the branch predictor (*BP*) updates and every time the BP of the undead core gets updated, a hint will be sent through the queue. In the animator core side, the default BP – for EV4 – is a simple bimodal predictor. We firstly 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. During our design space exploration, the size of these structures will be determined – Section 5.2. As mentioned earlier, we introduced release window size to get 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 the perfect timing of the hints. In such a scenario, for a subset of instructions, the tournament predictor can give an ad-

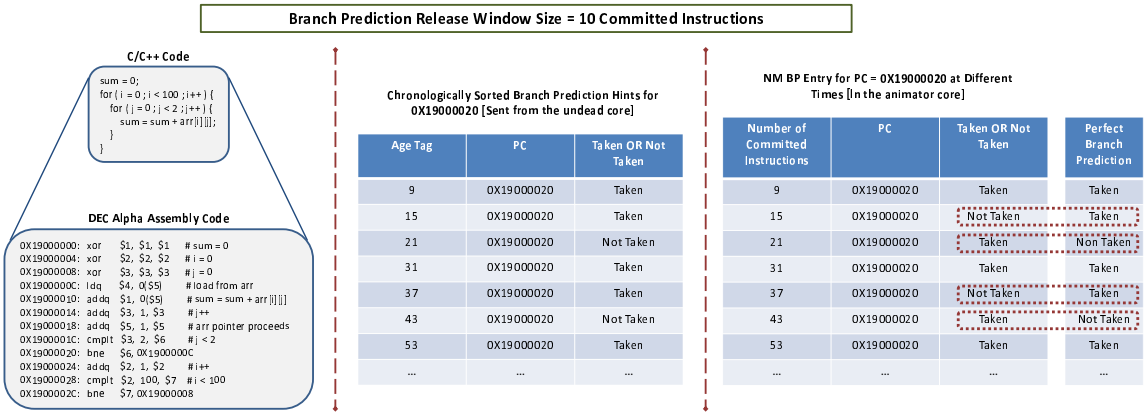


Figure 6: A code example in which the NM BP performs poorly and switching to the original BP of the animator core is required. The code simply calculates the summation of a 2D-array elements which are stored in a row-based format. It should be noted that the branch prediction release window size is normally set so that the branch prediction accuracy for the entire execution gets maximized. As can be seen, hints are received by the animator core at improper times, resulting in low branch prediction accuracy.

vantage to the original BP of the animator core to avoid any performance penalty. Having this in mind, Figure 6 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 – number of instructions in the loop body is less than BP release window size – with a low trip count. Switching to the original BP can enhance the overall branch prediction accuracy for this code region.

Another aspect of the NM dual core execution is the potential of hints on the speculative execution paths. If a speculative path turns to be a correct path, instructions on this path will eventually be committed and the corresponding hints will be sent to the animator core. On the other hand, for a wrong path, although sending hints can potentially accelerate the execution of speculative paths on the animator core, this acceleration can only decrease the efficiency of our hints for the correct paths. For instance, if the animator core executes a wrong path faster, it will bring more useless data to its local D-cache which causes prefetched data for non-speculative paths to be dropped out of D-cache. Therefore, it is clear that sending hints for speculative paths can merely hurt the performance of the NM system.

4.3 Reducing Communication Overheads

In order to reduce the queue size, communication traffic needs to be limited to more 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 the resource contention on the animator core side. For this purpose, these two CAMs keep track of the last N – number of CAM entries – committed load/store addresses in the undead core. In addition to sending less number of hints, queue size can be reduced by sending less bits per hint. Saving on the number of bits can be done in several ways: sending only the block related bits of address for I-cache and D-cache hints, ignoring hints on the speculative paths, and for branch prediction hints, only sending lower bits of the PC that are used for updating branch history table of the NM BP.

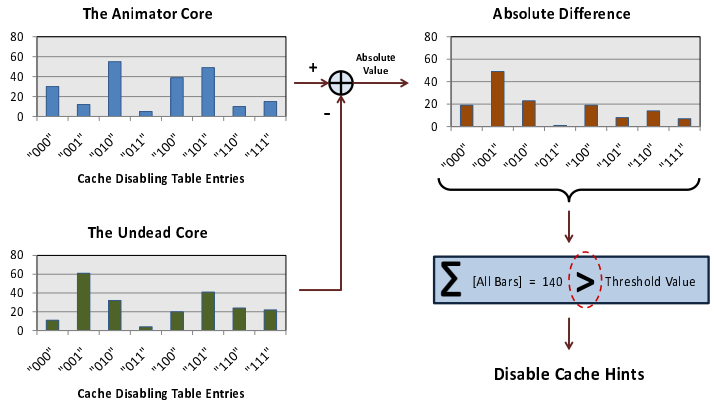
Given a design with multiple communication queues, the undead core stalls when at least one queue is full and it wants to insert a new entry to that queue. The other queues that are not full during these stalls remain underutilized; thus, using a single aggregated queue guarantees a higher utilization, which reduces the area overhead, number of stalls, and overheads of interconnection wires. On the other hand, since a single queue is used, multiple entries might need to be sent to or received from the queue at the same cycle. This can be solved by grouping together several hints with the same age tag and sending them as a single packet over the queue. This requires a small buffer in the hint distribution unit to handle the case that hints have non-identical release windows sizes.

4.4 Hint Disabling Mechanisms

Hints can be disabled when they are no longer beneficial for the animator core. This might happen because of several reasons. First, the program execution on the undead core gets off the correct execution path due to the destructive impact of defects. Second, in certain phases of the program, performance of the animator core might be close to its ideal case, attenuating the value of hints. Lastly, at certain parts of the program, due to the intertwined behavior of the NM system, the animator core might not be able to get ahead of the undead core. In all these scenarios, hint disabling helps in four ways:

- It avoids occupying resources of the animator core with ineffective hints that does not buy any performance benefit.
- The queue fills up less often which means less number of stalls for the undead core.
- Disabling hint gathering and distribution saves power and energy in both sides.
- It serves as an indicator of when the undead core has strayed far from the correct path of execution (i.e., when hints are frequently disabled) and resynchronization is required.

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 disabling 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 disabling should happen. In addition, when a hint gets disabled, that hint remains disabled during a time period called the back-off period. More precisely, the cache fingerprint unit retains two tables for keeping track of non-speculative I-cache and D-cache accesses in the last disabling time interval. Figure 7(a) illustrates an example of cache disabling. Considering D-cache hints, the corresponding table has only several entries – 8 entries in our example – and each entry will be incremented for a commit-



(a) Disabling cache hints

Resolved Branch Result	T	N	T	T	T	N	N	N
NM Branch Predictor	T	T	T	N	N	N	N	N
Original Branch Predictor of the Animator Core	T	T	N	N	T	T	T	N
Instantaneous Score	0	0	1	0	-1	1	1	0
Cumulative Score	0	0	1	1	0	1	2	2

Disabling Time Interval

Cumulative Score = 2 < Threshold Value

Disable Branch Prediction Hints

(b) Disabling branch prediction hints

Figure 7: Two high-level examples of cache and branch prediction hint disabling mechanisms. Here, values on the X-axes of the plots correspond to eight entries of the cache disabling table.

ted load/store, whenever the LSBs of the address match the rank order of that entry. Therefore, the cache disabling table maintains a high-level distribution of addresses that are accessed during the last interval. At the end of each interval, the table contents will be sent over the queue to the animator core and entries will be cleared for the next interval. Given a similar cache access distribution at the animator core's side, for evaluating similarity between two distributions, $(V_1, V_2, ..., V_{16})$ for the undead core and $(S_1, S_2, ..., S_{16})$ for the animator core, we calculate $K = \sum_{i=1}^{16} |S_i - V_i|$. Then, if K (140 in our example) is less than a pre-specified threshold, a signal will be sent to the undead core to stop gathering that particular hint for the back-off period.

Disabling branch prediction hints can solely be done by the animator core. Apart from prioritizing the original BP of the animator core for a subset of PCs, the NM BP can be also employed for global disabling of the branch prediction hints. For this purpose, we continuously monitor the performance of the NM BP and if this performance – compared to the original BP – is worse than a pre-specified threshold for the last disabling time interval, we disable branch prediction hints. As Figure 7(b) depicts, for branch prediction hint disabling, we use a score-based scheme with a single counter. For every branch that the original and NM BPs either both correctly predict or both mispredict 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 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 score counter (2 in our example) is less than a certain threshold, the branch prediction hints will be disabled for the back-off period. For performing infrequent disabling-related computations, we add a 4-bit ALU to the hint disabling unit.

4.5 Resynchronization

Since the undead core might get off the correct execution path, a mechanism is required to take it back to a valid architectural state. In order to do so, we use resynchronization between the two cores during which the animator core's PC and architectural register values get copied to the undead core. According to [27], for a modern processor, the process of copying PC and register values between cores takes on the order of 100 cycles. Moreover, all instructions in the undead core's pipelines are squashed, the rename table is reset, and the D-cache content is also invalidated for "resynchronizing" the memory state.

Resynchronization should happen when the undead core gets off the correct execution path and it can no longer provide useful hints for the animator core. The simplest policy is to resynchronize every N committed instructions where N is a constant number like 100K. However, as we will show in Section 5.2, a more dynamic resynchronization policy can achieve a higher overall speed-up for the NM system. We take advantage of the hint disabling information to identify when resynchronization should happen. An aggressive policy is to resynchronize every time a hint gets disabled. However, such a policy results in too many resynchronizations in a short time which clearly reduces the efficiency of our scheme. Another potential policy is to resynchronize only if at some point in time all or at least two of the hints get disabled. Later in Section 5.2, we will compare some of these potential resynchronization policies.

4.6 NM Design for CMP Systems

So far, we described the NM heterogeneous coupled core execution approach and its architectural details. Here, NM for CMP systems will be discussed. Figure 8 illustrates the NM design for a 16-core CMP 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 building block. Although a single animator core might be shared among more cores, it introduces long interconnection wires that should travel from one corner of the die to another.

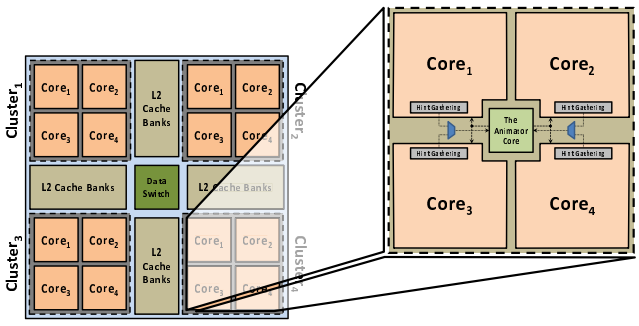


Figure 8: The high-level NM design for a large CMP system with 16 cores, modeled after the Sun Rock processor, which has 4 cores per cluster. The details of NM core coupling can be found in Figure 4.

Therefore, given the low area overhead of NM for a 4-core CMP (5.3% as will be discussed in Section 5.2), the proposed building block preserves design scalability. On the other hand, 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. One possibility is to use the animator cores to exploit Speculative Method-Level Parallelism by spawning an extra thread and moving it to the animator core to execute the method call. The original thread executes the code that follows the method's return by leveraging a return value predictor. This is based on the observation that inter-method dependency violations are infrequent. However, evaluation of the latter is beyond the scope of this work.

For a heterogeneous CMP system, the problem is slightly more difficult due to the inherent diversity of the cores. Therefore, sharing an animator core between multiple cores might not be possible since those cores have different computational capabilities. A potential solution is to partition the CMP system to groups of cores in which each group contains cores with similar characteristics and performance. Therefore, each group can share an animator core with different specifications. An alternative is to partition the cores to groups such that in each group, we have several large cores and a small core – all from the original set of heterogeneous cores. 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 in its own group. These dual purpose cores are a suitable fit for many heterogeneous CMP systems that come with a bunch of simpler cores such as the IBM Cell processor.

In our design, since the animator core is shared among multiple cores, it is reasonable to shift the overheads to the animator core side to avoid replicating of the same module in the baseline cores. For instance, most of the similarity matching structures for hint disabling are located on the animator core side. Furthermore, since the undead core runs significantly ahead of the animator core in the program stream, the communication queue should also be closer to the animator core to reduce the timing overhead of accessing the queue and checking the age tags. Finally, disabling hints, when they are no longer beneficial, allows the undead core to avoid gathering and sending the hints which saves power/energy on both sides.

5. EVALUATION

In this section, we describe experiments performed to quantify the potential of NM in enhancing the system throughput.

5.1 Experimental Methodology

In order to model NM's heterogeneous coupled core execution, we heavily modified SimAlpha, a validated cycle accurate microarchitectural simulator based on SimpleScalar [5]. We run two different versions of the simulator, implementing the undead and animator cores, and use inter process communication (*IPC*) to model the information flow between two cores (e.g., L2 warm-up, hints, and cache fingerprints). As mentioned earlier, a 6-issue OoO EV6 and a 2-issue OoO EV4 are chosen as our baseline and animator cores, respectively. The configuration of these two coupled cores and the memory system is summarized in Table 1. We simulate the SPEC-CPU-2K benchmark suite cross-compiled for DEC Alpha and fast-forwarded to an early SimPoint [31].

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 Monte Carlo simulation, a microarchitectural structure is selected and a random single stuck-at fault is injected into the timing simulator. Table 2 summarizes the fault locations used in our experiments. Since every transistor has the same probability of being defective, hard-fault injections should be distributed across microarchitectural structures in proportion to their area. Therefore, for each fault injection experiment, we inject 5000 hard-faults while artificially prioritizing structures that have larger area. These stuck-at faults are injected one by one in the course of each individual experiment. As a result, at any point in time, there is a single stuck-at fault in the undead core. Given an operational frequency of 600MHz [22] for EV6 in $0.35\mu m$, scaling to a $90nm$ technology node would result in a frequency of $2.3GHz$ at 1.2V. This frequency is a pessimistic value for the animator core and NM can clearly achieve even better overall performance if the animator core were allowed to operate at a higher frequency. Nevertheless, since the amount of work per pipeline stage remains relatively consistent across Alpha microprocessor generations [22], for a given supply voltage level and a technology node, the peak operational frequency of these different cores are essentially the same.

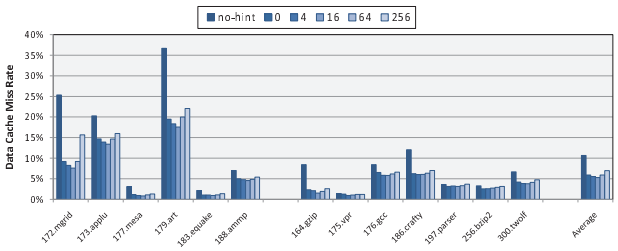
Dynamic power consumption for both cores is evaluated using Wattch [13] and leakage power is evaluated with HotLeakage [39]. Area for our EV6-like core – excluding the I/O pads, interconnection wires, the bus-interface unit, L2 cache, and control logic – is derived from [22]. In order to derive the area for the animator core,

Table 1: The target NM system configuration.

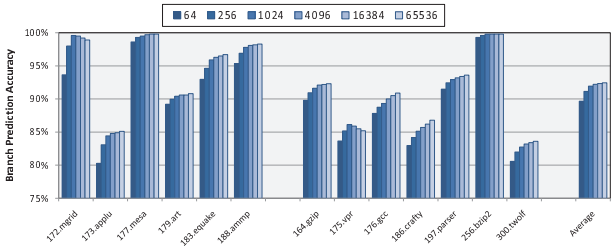
Parameter	The animator core	A baseline core
Fetch/issue/commit width	2 per cycle	6 per cycle
Reorder buffer entries	32	128
Load/store queue entries	8/8	32/32
Issue queue entries	16	64
Instruction fetch queue	8 entries	32 entries
Branch predictor	tournament (bimodal + NM BP)	tournament (bimodal + 2-level)
Branch target buffer size	256 entries, direct-map	1024 entries, 2-way
Branch history table	1024 entries	4096 entries
Return address stack	-	32 entries
L1 data cache	8KB direct-map, 3 cycles latency, 2 ports	64KB, 4-way, 5 cycles latency, 4 ports
L1 instr. cache	4KB direct-map, 2 cycles latency, 2 ports	64KB, 4-way, 5 cycles latency, 1 port
L2 cache	2MB Unified, 8-way, 15 cycles latency	
Main memory	250 cycles access latency	

Table 2: Fault injection locations and their corresponding pipeline stages along with stage-level area break-down for EV6.

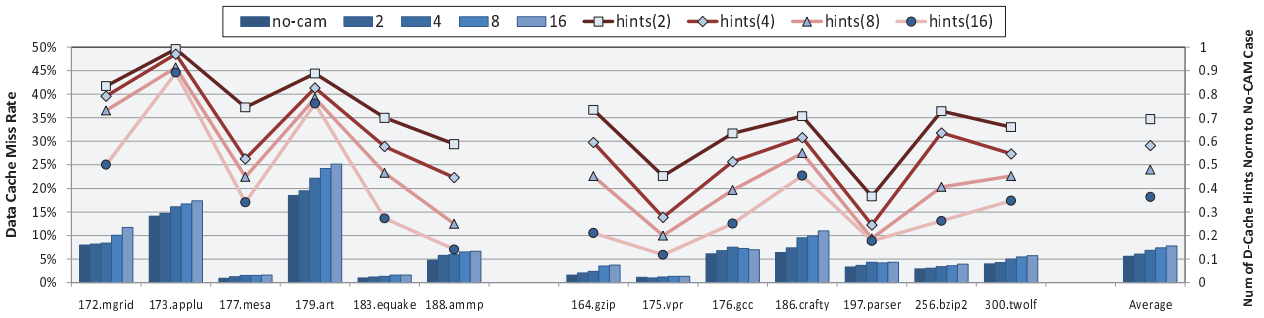
Pipeline Stage	Area Break-down	Fault Location
Fetch	14.3%	Program counter
		Branch target buffer
		Instruction fetch queue
Decode	15.6%	Input latch of decoder
Rename	5.1%	Rename alias table
Dispatch	24.1%	Integer register file
		Floating point register file
		Reorder buffer
Backend	40.8%	Integer ALU
		Integer multiplier
		Integer divider
		Floating point ALU
		Floating point multiplier
		Floating point divider
		Load/store queue



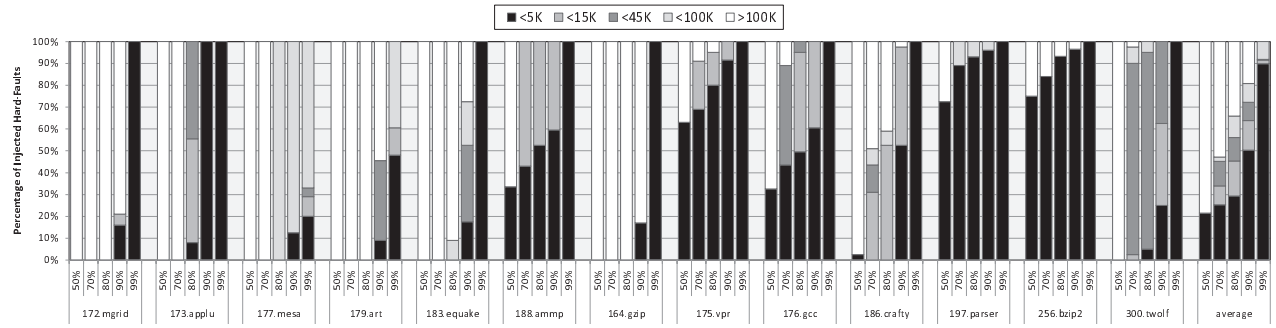
(a) Effect of the NM D-cache release window size on the data cache miss rate of the animator core.



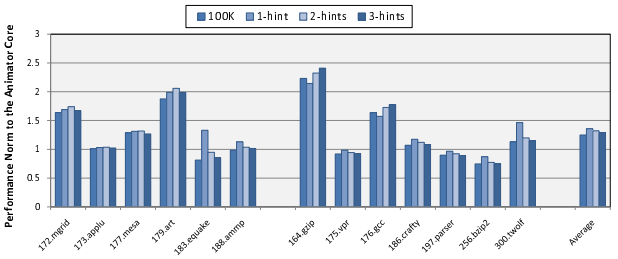
(b) Effect of the branch history table size of the NM BP on the overall branch prediction accuracy of the animator core.



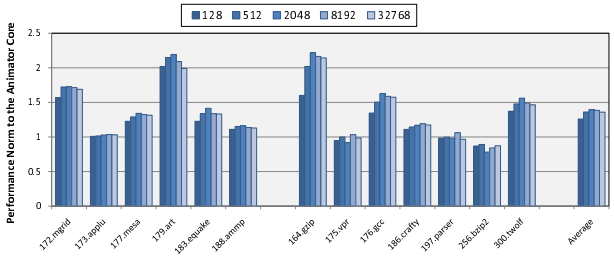
(c) Effect of CAM size that are used for reducing the number of D-cache hints – generated in the undead core – on the data cache miss rate of the animator core. Here, the lines show the number of data cache hints should be sent to the animator core per cycle, normalized to the the case without any CAM.



(d) Number of instructions committed in the animator core before the branch prediction hint is disabled for different pre-specified branch prediction hint disabling thresholds (i.e., 50%, 70%, 80%, 90%, and 99% similarities).



(e) Effect of different resynchronization policies on the overall speed-up of the NM coupled cores normalized to the performance of the baseline animator core.



(f) Effect of communication queue size on the overall speed-up of the NM coupled cores normalized to the performance of the baseline animator core.

Figure 9: Design space exploration for the NM system described in Table 1.

we start from the publicly available area break-down for the EV6 and resize every structure based on the size and number of ports. Furthermore, CACTI [26] is used to evaluate the delay, area, and power of the on-chip caches. Overheads for the SRAM memory structures that we have added to the design, such as the NM branch prediction table, are evaluated with the SRAM generator module provided by the 90*nm* Artisan Memory Compiler. Moreover, 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., MUXes, shift registers, and comparators). Finally, the area for interconnection wires between the coupled cores is estimated using the same methodology as in [23], with intermediate wiring pitch taken from the ITRS road map [19].

5.2 Experimental Results

In this section, we evaluate different aspects of the NM design

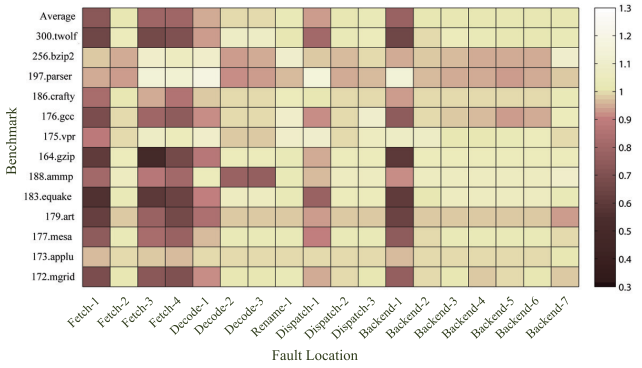


Figure 10: Variations in the speed-up of the animator core for different hard-fault locations across SPEC-CPU-2K benchmarks. To only highlight the impact of hard-fault locations, in each row, results are normalized to the average speed-up that can be achieved by the NM coupled cores for that particular benchmark.

such as design space, achievable speed-up in the presence of defects, performance impact of different hard-fault locations, area and power overheads, and finally throughput enhancement.

Design Space Exploration: Here, we fix the architectural parameters that are involved in the NM design. Since there is a variety of parameters (both hardware and policy), due to space considerations, we only present a subset of the exploration for parameters with the most interesting behaviors. During the exploration, we initially assign a nominal value to each of the parameters and as we select a proper value for each parameter, we use the updated value for the reminder of the experiments. Figure 9 depicts this design space exploration for a pruned set of NM parameters. In Figure 9(a), the release window size is varied between 0 to 256 committed instructions while monitoring the data cache miss rate of the animator core. As can be seen, there is an optimal window size (i.e., 16 committed instructions) that maximizes prefetching efficiency, given the variations in the number of committed instructions on the undead core. The D-cache miss rate, even before optimizing other parameters, is reduced from 10.7% to 5.3%. Figure 9(b) illustrates the effect of reducing the branch history table (*BHT*) size of the NM BP on the branch prediction accuracy of the animator core. To save area, we limit the BHT size to 1024 entries, causing less than 0.5% reduction in the achievable branch prediction accuracy.

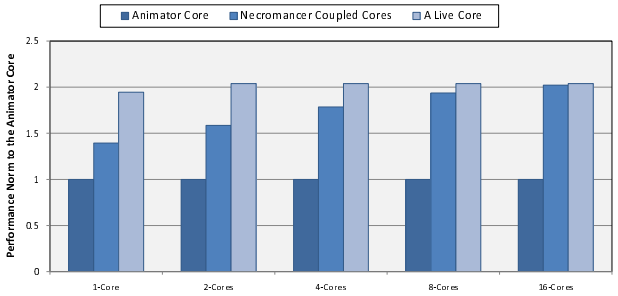
The size of the D-cache hint CAM is a double-edged sword and its impact on the D-cache miss rate and communication traffic is shown in Figure 9(c). Increasing the CAM size, reduces the communication traffic and queue size. However, this aggravates the efficiency of D-cache hints. The reason is that sending more up-to-date hints increases the likelihood that data is present in the local D-cache of the animator core when it is needed. Nevertheless, using a CAM with 2 entries can reduce the number of transmitted D-cache hints by more than 30% while affecting the D-cache miss rate by less than 0.5%. Next, Figure 9(d) illustrates the effect of varying the threshold for disabling branch prediction hints. For each injected hard-fault and benchmark, we record the number of instructions committed before the branch prediction hint is disabled. Results of this process are depicted for 5 different threshold values (i.e., 50%, 70%, 80%. 90%, and 99% similarities). For high similarity requirements, such as 99%, the branch prediction hints are mostly disabled even before 5K instruction are committed in the animator core. Consequently, we select 70% similarity so that the hint disabling does not occur too frequently while still receiving occasional feedback about the effectiveness of the hints during program execution.

Finally, Figures 9(e) and (f) show the impact of different resynchronization policies and communication queue sizes on the achievable speed-up by NM, respectively. In these two plots, speed-ups are normalized to the performance of a baseline animator core. We consider 4 candidates for the resynchronization policy, consisting of one static and 3 dynamic polices. For the static policy, resynchronization occurs periodically after committing 100K instructions while for the dynamic policies, the number of disabled hints determines whether resynchronization is required. Since we aggressively exploit the hints by rarely disabling them, the resynchronization policy that is invoked on the first disabled hint achieves a better speed-up. Finally, the sensitivity to the communication queue size is presented in Figure 9(f). Although it seems that a larger queue is always better, an extremely large queue enables the undead core to get too far ahead of the animator core, polluting the L2 cache with unprofitable prefetches.

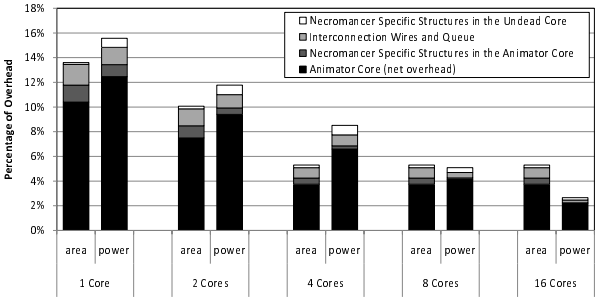
The values for the remaining parameters were identified in a similar fashion: I-cache release window size (4 committed instructions), branch prediction release window size (4 committed instructions), I-cache hint CAM size (2 entries), branch prediction hint disabling threshold (70% similarity), D-cache hint disabling threshold (70% similarity), I-cache hint disabling threshold (80% similarity), D-cache hint disabling table size (32 entries), and I-cache hint disabling table size (32 entries). Given these parameter values, on average, NM can achieve 39.5% speed-up over the baseline animator core. In our simulation, we set the queue delay to 15 cycles – same as L2 cache; however, since the NM coupled core design is highly pipelined, it has a minimal sensitivity to the queue delay. For instance, even setting this delay to 45 cycles, only affects the final speed-up by less than 1%.

Performance Impact of Different Hard-Fault Locations: In order to highlight the impact of a fault location on the achievable speed-up by the NM system, Figure 10 depicts the performance breakdown results for the fault locations described in Table 2. Results in each row of this plot is normalized to the average speed-up that can be achieved by the NM coupled core 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, for a benchmark like 197.parser, reaction to defects can significantly differ from other benchmarks. We conclude two main points from this plot. First, on average, there are only a few fault locations that can drastically impact the NM speed-up gain. 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 having a more suitable workload assigned to the coupled cores.

Summary of Benefits and Overheads: Figure 11(a) demonstrates the amount of speed-up that can be achieved by the NM coupled cores for CMP systems with different numbers of cores. As can be seen, NM achieves a higher overall speed-up as the number of cores increases. 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. This is because NM achieves different speed-ups based on the defect type, location, and the workload running on the system. 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,



(a) Performance of the baseline animator core, NM coupled cores, and a live core normalized to the average performance of a baseline animator core. Due to the higher heterogeneity across the benchmarks for a CMP system with more cores, NM can achieve a higher overall speed-up.



(b) Break-down of NM area and power overheads for CMP systems with different numbers of cores. As can be seen, the overheads that are imposed by the the baseline animator core is typically the major component, which gets amortized as the number of cores grows.

Figure 11: Summary of benefits and overheads of our scheme for CMP systems with different number of cores.

there is more opportunity for NM to exploit. The speed-up evaluation was done by conducting a Monte Carlo simulation with 1000 iterations. In each iteration, we select one benchmark for each core, while allowing replication in the selected benchmarks.

Figure 11(b) shows the breakdown of area and power overheads for our scheme. Here, we assume a single core system has 2MB L2 while assuming 1MB shared L2 per core for CMP systems. 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, two points should be noted. First, based on our target defect rate, for CMPs with more than 4 cores, other animator cores remain disabled and do not contribute to the power consumption. Next, as the speed-up results show, for CMPs with less than 8 cores, the undead core remains ahead of the animator core and it needs to stall when the queue gets full. During stall times, the undead core does not consume dynamic power which is accounted for in the net overhead of the animator core – Figure 11(b).

Finally, as discussed earlier, based on the expected defect rate for current and near future CMOS technologies, on average one defect per five manufactured $100mm^2$ dies should be expected. 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.

6. RELATED WORK

Manufacturing defects can cause transistors in different parts of a microprocessor to get corrupted. Prior work on defect tolerance mostly focused on on-chip caches since there is less homogeneity in the non-cache parts of a core, making defect tolerance a more challenging issue. Typically, for high-end server systems designed with reliability as a first-order design constraint (e.g., HP Tandem NonStop [7], Teramac [15], and the IBM eServer zSeries [7]), coarse-grained replication has been employed [8, 33]. Configurable Isolation [2] is a high availability chip multiprocessor architecture for partitioning cores to multiple fault domains which allows independent redundant executions. However, dual and triple modular redundant systems incur significant overheads in terms of area and power which is not generally acceptable for mainstream computing. An easy solution is to disable the faulty cores – to avoid yield loss – which clearly causes a significant reduction in the system throughput and sale price [2]. This simple core disabling approach has been taken by microprocessor vendors, such as IBM, Intel, AMD, and Sun Microsystems, to maintain an acceptable level of manufacturing yield.

Core Cannibalization [29] and StageNet [17] suggest 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. Finer-grained redundancy maintenance has been used by Bulletproof [14] and sparing of array structures [11]. In the same vein, Shivakumar et. al. [32] proposed a method to disable non-functional microarchitectural components (e.g., execution units) and faulty entries in small array structures (e.g., register file). Rescue is mainly a microarchitectural design-for-test (*DFT*) technique which can map out faulty pipeline units that have spares [30]. However, as shown in [27], these schemes have a limited applicability due to the small amount of microarchitectural redundancy that exists in a modern high-performance processor.

Architectural Core Salvaging [27] is a high-level low-cost architectural proposals which uses thread migration between the cores to guarantee the correct execution. To avoid incorrect execution, for each instruction, it assesses whether the fault location might be exercised by the corresponding *opcode*. Thus, without using extra redundancy, it is only applicable to defects in about 10% of core area. DIVA [4] was proposed for dynamic verification of complex high-performance microprocessors. It employs a checker pipeline that re-runs the same instruction stream for ensuring correct program execution. Given the fact that DIVA is not a defect tolerant scheme, as shown in [4], a "catastrophic" core processor failure results in about 10X slow-down. Detour [25] is a completely software-based approach which leverages binary translation for handling defects in execution units and register files. Apart from limited defect types that can be handled, a binary translation layer cannot typically be applied to high-performance x86 cores [27].

7. CONCLUSION

Since manufacturing defects directly impact yield in nanoscale CMOS technologies, to maintain an acceptable level of manufacturing yield, these defects need to be addressed properly. Non-cache parts of a core are less structured and homogeneous; thus, tolerating defects in the general core area has remained a challenging problem. In this work, we presented Necromancer, an architectural scheme to enhance the system throughput by exploiting dead cores. Although a dead core cannot be trusted to perform program execu-

tion, for most defect incidences, its execution flow – when starting from a valid architectural state – coarsely matches the intact program behavior for a long time period. Hence, Necromancer does not rely on correct program execution on a dead core; instead, it only expects this undead core to generate effective execution hints to accelerate the animator core. In order to increase Necromancer efficacy, we use microarchitectural techniques to provide intrinsically robust hints, effective hint disabling, and dynamic inter-core state resynchronization. For a 4-core CMP system, on average, our approach enables the coupled core to achieve 87.6% of the performance of a live core. This defect tolerance and throughput enhancement comes at modest area and power overheads of 5.3% and 8.5%, respectively. We believe NM is a valuable and low-cost solution for tolerating manufacturing defects and improving the throughput of the current and near future mainstream CMP systems.

8. ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments and suggestions. This research was supported by National Science Foundation grants CCF-0916689 and CCF-0347411 and by ARM Limited.

9. REFERENCES

[1] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. González. Low vccmin fault-tolerant cache with highly predictable performance. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, page To appear, 2009.

[2] 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.

[3] A. Ansari, S. Gupta, S. Feng, and S. Mahlke. Zerehcache: Armoring cache architectures in high defect density technologies. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009.

[4] 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.

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

[6] 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.

[7] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two sytems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[8] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[9] K. Bernstein. Nano-meter scale cmos devices (tutorial presentation), 2004.

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

[11] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, page 51, 2004.

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

[13] 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.

[14] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: A defect-tolerant CMP switch architecture. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 3–14, Feb. 2006.

[15] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the teramac custom computer. In *Proc. of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 116–123, 1997.

[16] 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.

[17] S. Gupta, S. Feng, A. Ansari, J. 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.

[18] T. Higashiki. Status and future lithography for sub hp32nm device. In *2009 Lithography Workshop*, 2009.

[19] ITRS. International technology roadmap for semiconductors 2008, 2008. http://www.itrs.net/.

[20] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[21] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, 2007.

[22] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[23] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-core chip multiprocessing. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 195–206, 2004.

[24] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 105–116, 2009.

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

[26] 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.

[27] 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*, page To Appear, June 2009.

[28] 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.

[29] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processor in the presence of hard faults. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[30] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 160–171, 2005.

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[32] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of the 2003 International Conference on Computer Design*, page 481, Oct. 2003.

[33] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999.

[34] E. Sperling. Turn down the heat...please, 2006. http://www.edn.com/article/CA6350202.html.

[35] 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.

[36] N. J. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–65, 2003.

[37] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*, page 61, June 2004.

[38] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off cache capacity for reliability to enable low voltage operation. *Proc. of the 35th Annual International Symposium on Computer Architecture*, 0:203–214, 2008.

[39] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. *ACM SIGARCH Computer Architecture News*, 31(2):136–146, 2003.

[40] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, 2005.

[41] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.