Reducing Overhead for Soft Error Coverage in High Availability Systems

Nidhi Aggarwal, Norman P. Jouppi, James E. Smith
Parthasarathy Ranganathan and Kewal K. Saluja

Abstract—**High reliability/availability systems typically use redundant computation and components to achieve detection, isolation and recovery from faults. Chip multiprocessors (CMPs) incorporate multiple identical components on a chip to provide high performance/watt. These identical components can be used in a redundant configuration to build cost-effective high availability systems. Current high availability systems like NonStop and Stratus replicate all components of the system including cores, system circuitry and memory. However, it would likely be infeasible to replicate all the components of a system in low cost commodity CMP-based high availability system. In this paper, we focus on low overhead techniques to detect logic soft errors in high availability systems. We specifically focus on reducing the overhead of replicating 1) memory because it is the most expensive component of a system and 2) cores because the number of cores in a system affects its throughput significantly. Our memory duplication overhead reduction technique is based on the intuition that memory storage itself is protected with ECC and errors in logic (cores, on-chip circuitry) can not propagate to pages that are never written. Therefore, we propose limiting the replication to memory pages that are written. For reducing the core duplication we leverage the fact that only certain applications are availability sensitive in a general purpose environment. We propose to selectively use redundant computation for only availability-sensitive applications. Our techniques can provide significant reduction in overheads for a minor performance penalty. We believe that the combination of these two techniques will improve the economic attractiveness of high availability systems built using commodity CMPs.**

***Index Terms*— Availability, Fault tolerance, Reliability, System Recovery, Memory duplication.**

I. Introduction

High availability systems typically rely on multiple redundant components and computation to detect, isolate and recover from errors. There are several ways in which the system components can be configured redundantly to achieve these essential conditions for fault tolerance. For example, the IBM zSeries systems [1] use fine grained replication and voting at each component with custom designed tightly lock-stepped processor pipelines whose outputs are compared before reaching the memory hierarchy. On the other hand, the NonStop [2] and Stratus systems [3] use coarse grained replication using commodity components where the processor, cache, and memory are replicated and the outputs are compared at the I/O hub.

As error rates increase in successive processor generations [4], similar techniques will likely need to be adopted in commodity high volume servers. Chip multiprocessors (a.k.a. multicores) have multiple identical components and are an ideal building block for commodity high availability systems. However, cost will be a major determinant of the types of techniques that are suitable for use in commodity systems. Further, not all the applications in a general purpose volume server environment will require high availability. Therefore, techniques that require custom design for high availability and impose the 100-200% performance overhead for Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) on all applications are likely to be prohibitive. In this paper, we focus on systems that can use a commodity chip multiprocessor to build a high availability system, and explore the potential for reducing the overheads of 100% duplication.

Manuscript submitted December 21, 2007. This work was done at University of Wisconsin-Madison. N. Aggarwal, J. E. Smith, and K.K. Saluja are with University of Wisconsin-Madison, Madison, WI 53715 USA (e-mail: naggarwal@wisc.edu, jes@ece.wisc.edu, saluja@engr.wisc.edu).
N. P. Jouppi, and P. Ranganathan are with HP, Palo Alto, CA 94304 USA (email: norm.jouppi@hp.com , partha.ranganathan@hp.com)

II. Replication Overheads

In traditional fault tolerant systems, the components that need to be replicated depend on the granularity of error containment [5] in the system based on how far outside the core the error is allowed to propagate. Typically, the components that are within the error containment boundary are replicated and their outputs are compared before they can be released to components outside the granularity of the error containment. For example, if errors are contained to the processor core, then either the processor pipelines [1], threads [8] or processors [9] are replicated. In cache containment, both the processor and their caches are replicated [10, 11]. Both processor level and cache-level error containment require complex modifications to the critical chip components [5]. For example, the IBM zSeries systems [1] replicate the processor pipeline and operate them in tight lock-step with a common clock. Supporting tight lock-stepping in future systems is becoming increasingly difficult (if not impossible) because of the complexity of synchronization in the presence of local error recovery (e.g. ECC, cache retries), power management using variable frequencies, etc. [2, 7]. Software transparent cache containment systems [11] require frequent checkpoints and substantial changes to the cache coherence controller – a difficult component to design and verify.

Permitting errors to propagate to memory can enable the use of commodity CMPs because it does not require modifications to the cores or the coherence protocol. The processors operate relatively independently in loose lock-step [2, 3] and can

tolerate variations due to local error recovery. The results are compared at the I/O level. However, loose-lockstepped systems require that the processor, cache, and *memory* be replicated. Because of the difficulty in supporting tight lock-stepping, the complexity of cache containment approaches, and the advantages of loose-lockstepping, we focus on loose-lockstepped systems.

We divide the replication overheads in loose lock-stepped high availability systems into two classes – logic replication and storage replication. Examples of logic overhead include core replication, cache controller, and memory controller replication. Examples of storage replication include cache replication and memory array replication. For logic, reducing the core replication overhead is most important because the core occupies the maximum logic area, and dedicating a core for redundant computation affects the throughput of the system significantly. For storage, reducing the memory replication overhead is important because memory is the most expensive component in server systems [5], and future workloads are likely to have high memory footprints (e.g. databases, enterprise resource planning applications and memory intensive multimedia desktop applications).

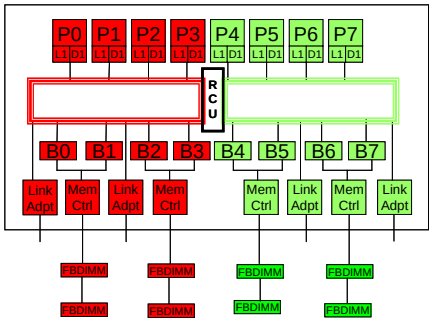


Figure 1. CMP based high availability systems. RCU is a self-checked ring configuration unit [13] to partition the interconnect.

The unit of replication, isolation and recovery in traditional loose-lockstepped fault tolerant systems like NonStop and Stratus is an entire processor board. This unit is too coarse grained for future CMPs where multiple components are integrated on a single socket. We therefore focus on low cost fault tolerant systems that can use commodity CMPs as building blocks. In order to implement a high availability system using a commodity CMP where resources like caches and memory controllers are shared, it must be possible to have complete isolation between the components running the redundant processes. Configurable isolation [12] (Figure 1) is a recent proposal to dynamically partition the shared resources in a CMP into different colors that are fully isolated (red and green in Figure 1, black and grey in black and white print). These different colors can now be used to map redundant computation to achieve effectively 100% fault detection. Traditional backward and forward error recovery techniques can then be used in a CMP based system. A system with configurable isolation can be configured as a high availability system with I/O level voting and loose-lockstepped processors similar to the NonStop DMR system.

CMPs offer new opportunities for optimizations to reduce overheads compared to fully replicating memory and processor cores for applications in commodity high availability systems. In this paper we discuss techniques that can leverage software management and small hardware hooks in a commodity CMP to reduce the overheads.

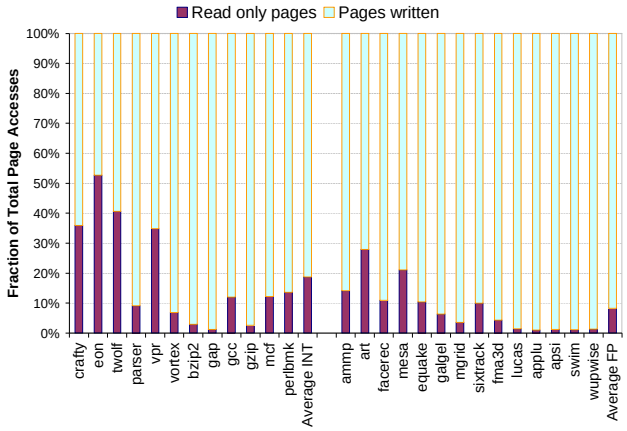
III. MEMORY OVERHEAD REDUCTION

Memory has traditionally been susceptible to soft errors and therefore techniques like extensive ECC, chip kill, DIMM sparing etc. have been incorporated even in commodity systems to make the memory system robust. Also, an important class of hard memory errors is fail stop. For example, if an entire memory module fails then it doesn't respond. However, current memory system designs are susceptible to errors that originate in the processor cores, caches or memory controllers and are propagated to it. These faults are undetectable through memory system fault tolerance techniques like ECC which can detect and correct local errors. We therefore focus on devising techniques for checking all errors that either originate or are propagated to memory and isolating them at a page level (for software level management) - but without using the standard technique of fully duplicating memory. Once an error is detected, we envision that the system can roll back to a fault free state by using previously stored checkpoints by using techniques similar to current high availability systems.

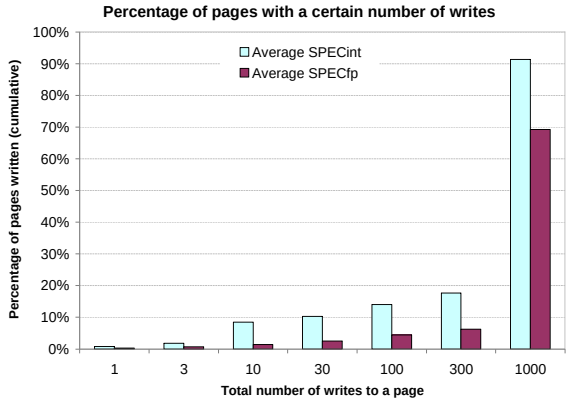
We make the observation that if an application does not modify a page then the errors that originate in the core or caches can not propagate to it. Therefore, read only pages need not be duplicated. A CMP-based system has the unique property that components can be shared dynamically. Using a shared memory path and system software support for modified page tables, all the replicas can be directed to a single copy of the read only pages. Using full system simulation for SPEC CPU2000 benchmarks, we found that on average, 18% of the pages in SPECint and 8% in SPECfp are read-only (figure 2(a)) and don't need to be duplicated. While not insignificant, not duplicating read only pages alone is not a big win.

To reduce the memory duplication overhead further, we observed that many pages, although not read-only, are written seldom (perhaps only once). If a page is written, but will not be written again for a long time interval (or never), then at a relatively low cost, the page can be "converted" to a read-only page. This conversion can be done by having system software compare the duplicated copies, and if they are the same (which will be the case unless there has been an error), one of the copies can be discarded while the other is marked as read-only. On average, for SPECint, 91% of pages written have fewer than 1000 writes and 18% have fewer than 300 writes (figure 2(b)). For SPECfp 69% of pages written have fewer than 1000 writes and 7% had fewer than 300 writes (figure 2(b)). This suggests that a fairly large number of pages are seldom written. Also, we observed that for SPEC CPU2000 the average time interval between writes is less than 10 cycles for 40% of pages written. Since the average interval between writes is small; this suggests that the writes to a page are typically clustered. Thus in these applications a large number of pages are written fewer than 1000 times and these writes

typically happen in quick succession - such pages only need to be duplicated during the time they are being actively written (i.e., are a part of the write working set of the application).



(a)



(b)

Figure 2. (a) Percentage of page access types for SPEC CPU2000 benchmarks (b) Histogram of numbers of writes to pages; cutoff at 1000.

However, there is a wide variation between the size of the write set of benchmarks; therefore, the amount of memory duplication required depends on the application behavior, which can be monitored at runtime. We propose a novel technique to dynamically adjust memory duplication at runtime by using a cache of frequently written pages – *the duplication cache*. Conceptually, the duplication cache is a region of real main memory managed by system software or replica manager. The duplication cache stores duplicates of the recently written pages. Initially all pages are marked read only. When a page is written for the first time, there is a page fault to system software and a duplicate page is created in the cache. When adding another page to the cache will exceed its capacity, then the Least Recently Written (LRW) page in the duplication cache is reclaimed by comparing it with its replica in main memory. If there is no mismatch (as will be the case unless there was an error), then the duplicate page is evicted from the cache, the page permissions for the replica in main memory are set to read only, and space is freed for the new duplicate. By choosing an appropriate size of the duplication cache, the amount of memory duplicated can be restricted to roughly the current write set of the application. The replica manager keeps track of the percentage of physical memory allocated to the duplication cache and to main memory using software counters.

In case there is a mismatch between the compared pages, then the system has detected an error before it reached the I/O voter system. In this case, the system software initiates the same recovery mechanism that is initiated when an error is detected at the I/O system. Typically, the recovery mechanism is to roll back execution to a previously stored checkpoint and restart execution from the checkpoint. The duplication cache does not interfere with the recovery mechanism.

A. Duplication Cache Implementation

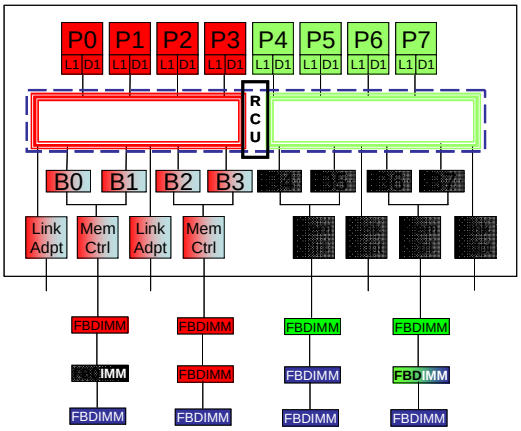


Figure 3. Physical design of the configurable memory duplication - green and red are partitioned components and blue represents shared data/components.

One of the attractive features of the proposed duplication cache is that it is simple to implement in CMP based high availability systems. All the mechanisms used are well understood by current OSes for optimizations such as Copy on Write (CoW) that are used to increase the memory system utilization and overall system performance. We outline the changes required in a CMP based high availability system to implement a duplication cache.

Figure 3 shows a detailed physical design of a CMP based high availability system, similar to the one proposed in [13], with a duplication cache implementation. In [13] the RCU either partitions the system into multiple domains or allows unconstrained sharing. We modify the Ring Configuration Unit to enable it to either a) allow sharing for read only data or b) partition the interconnect for writes. Although we show a ring based system in this paper, the techniques are also applicable to other types of interconnects.

Reads to non-duplicated pages carry a special 'read only' bit identifying them to the RCU. The RCU allows these reads to non-duplicated pages to travel on the ring unconstrained to any memory controller in the system – logically shown as the dotted/blue ring line in the figure. However, when not running

in shared mode writes are routed only within their own color (partition) for fault isolation purposes. As shown in Figure 3, the 'read only' (blue or dark grey) and write pages (red or green or black and grey in black and white print) can be mixed on a single DIMM because a replica of all the information in a single DIMM exists somewhere in the system (either on disk, a checkpoint, or the duplicated page). The only requirement is that the duplicate cache and the main memory replica should not be on the same DIMM.

Note that a duplication cache is not possible in traditional board level redundant high availability systems like NonStop and Stratus because there is no means to share memory. In this sense, CMP based systems are ideally suited for such overhead reducing optimizations.

B. Duplication Cache Evaluation

We evaluated the memory-sharing optimizations using a full system x86/x86-64 simulator (similar to AMD SIMnow) [19] that can boot an unmodified Windows or Linux OS and execute complex applications.

Memory size	Benchmarks (resident set size)
Individual benchmarks	
256 MB	INT - gcc(154), mc(190), gap(192), gzip (180), bzip2 (185), perlbmk (146)
	FP - wupwise(176), swim(191), applu(181), apsi(191)
192 MB	INT - vortex(72)
	FP - lucas(142), fma3d(103)
128 MB	INT - vpr(50), parser(37)
	FP - mgrid (56), galgel (63), equake (49), sixtrack (26)
96 MB	INT - crafty(2), eon(0.6), twolf(3.4)
	FP - mesa(9.4), art(3.7), facerec(16), ammp(26)
Workload consolidation	
2046 MB (very large)	gzip, mcf, gap, bzip2, swim, apsi, applu, wupwise
1536 MB (large)	lucas, fma3d, vortex, perlbmk, apsi, swim, crafty, twolf
1024 MB (medium)	mgrid, galgel, equake, sixtrack, vpr, parser, vortex, fma3d
768 MB (low)	mesa, art, facerec, ammp, crafty, eon, twolf, sixtrack

Figure 4. Benchmark and workload description

We first simulated all the benchmarks from the SPEC CPU2000 suite individually and then combined multiple benchmarks to simulate workloads. We divided the individual benchmarks into four categories based on their resident set size in memory and chose a base memory configuration for each category (figure 4). The four base memory sizes were – 96 MB, 128 MB, 192 MB and 256 MB. For the combined workload experiments we constructed four workloads - very large memory, large memory, medium memory and low memory - using 8 benchmarks for each workload. We simulated duplication cache sizes of 10%, 20%, 30% and 40% of each base memory size. For example, if the base memory size is 128 MB for vpr and the duplicate cache is 10% (12.8 MB), then the total memory for vpr simulation is 140.8 MB. The percentage of memory duplication (size of duplication cache) affects the number of pages that are evicted from the duplication cache and made read only (*page evictions*) and the number of pages that were evicted earlier but are written to again and thus reenter the duplication cache (*page re-entries*).

Performance overhead modeling: The number of page evictions and reentries that a particular size of the duplication cache entails affects the performance overhead imposed by duplication cache. We developed a simulation-based performance overhead model to characterize the overheads. The number of overhead cycles can be calculated as:

Overhead cycles = Number of page evictions * Cost of comparing one page + Number of pages written * (Cost of page copy + Cost of page protection fault) + Number of page reentries * (Cost of page copy + Cost of page protection fault)

We measured the number of page evictions, number of unique pages written and the number of page re-entries for each cache size for all the benchmarks simulated. We estimated the cost of comparing and copying one page of 4KB using published results for STREAM benchmark [20] for 64-bit commodity processors - Itanium, Opteron, and G5 systems using standard operating systems. The copy bandwidth reported was 3012 MB/s and the sum or compare bandwidth was 2577 MB/s. The page faults that duplication cache generates are *minor page faults* and do not impose the substantial overhead of a conventional page fault that brings a page to disk. This fault only requires a trap to the replica manager to perform the page copy and then one memory access to mark the page writeable. We assume a conservative 1000 cycles as the cost of a minor page fault. We also performed sensitivity studies with the page fault cost of 2000 and 500 cycles and observed that average change in overhead was less than 0.5%.

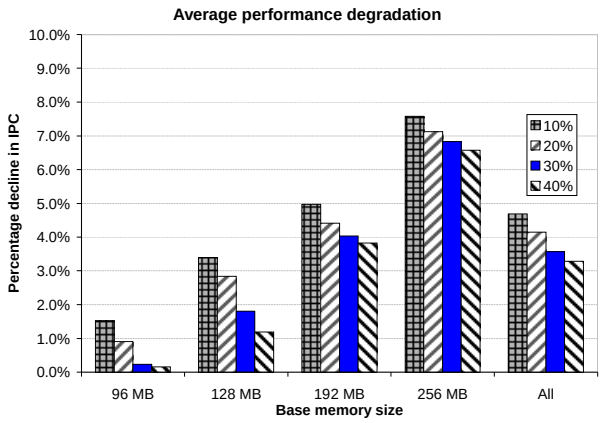


Figure 5. Performance overhead (individual benchmarks)

Figure 5 shows the average performance overhead for benchmarks in the four base memory categories for various sizes of the duplication cache. The average performance overhead across all categories is ~5% for a 10% duplication cache and ~3.5% for a 40% duplication cache. As expected the benchmarks with large memory footprints experience more performance overhead, ranging from 6.8% to 7.6%.

Figure 6 shows the average performance overheads experienced by our workload consolidation benchmarks. The performance overhead ranges from around 1% for a low memory workload to around 9% for a very large memory workload with 10% memory duplication. The results show that memory duplication can enable low cost soft error detection without a significant decrease in performance.

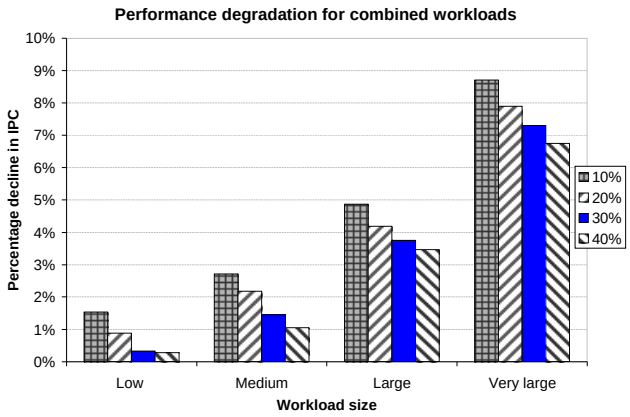


Figure 6. Performance overhead (combined workloads)

IV. CORE OVERHEAD REDUCTION

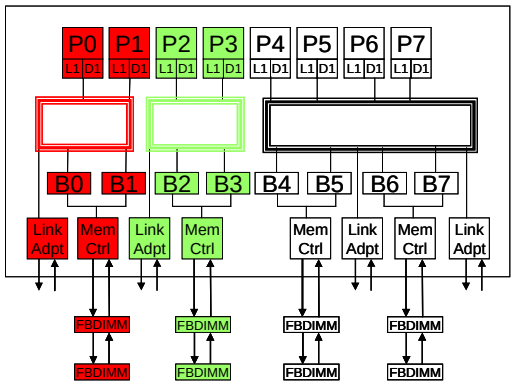


Figure 7. System architecture – Red/black and green/grey resources run redundant computation and white resources are used non-redundantly.

For core duplication, we first observe that future general purpose commodity CMP-based systems are likely to be used for a diverse mix of applications, in contrast to current high availability systems that typically run a limited set of mission critical applications. In such a general purpose environment, very high availability will not be required by all applications or at all times by a given application. For example, a DVD player application may not require higher availability than that natively provided by future CMPs; however, home banking applications or an e-commerce web server may require high availability. Also, in a consolidated server environment there might be different service level agreements for availability for different applications. Running applications that do not require high availability in a redundant configuration wastes power and can also reduce performance by consuming an extra share of the available CMP resources.

We propose using a system with configurable isolation to provide ***configurable redundancy*** for only the applications that require high availability. A system with configurable isolation can be set up to run certain applications redundantly and the rest of the resources can be used in a non-redundant configuration to reduce power usage (by shutting off cores) or to improve the throughput of the system. A simple software interface can inform the replica manager of the availability requirements of the application. The replica manger can create and manage redundant threads only for the availability sensitive applications so that the cost of redundant execution is paid for only the applications that need it.

Figure 7 shows an example 8 core system where the system has three partitions. Using RCUs and OS-assisted memory partitioning P0-P1 and P2-P3 are partitioned and mapped to different colors to run redundant processes in a DMR configuration. But processors P4-P7 constitute a single partition shared partition (in effect a four core CMP) that can be used to run non-availability sensitive applications independently. Note that although the redundant and non-redundant domains exist on the same chip, they do not share any vulnerable components and errors cannot propagate from the low-availability application domain to the high availability application domain.

Configurable redundancy can provide benefits in terms of power, latency, and throughput but it is bimodal in the sense that it either runs the applications in a redundant configuration or not. It does not take into account the temporal variations that might occur in either the availability required or the resources available. The configurable isolation system can also be used to provide ***adaptive redundancy*** based on these temporal variations. Adaptive redundancy can be selectively turned on based on the current operating conditions and application characteristics. Below we list a few possible factors that could control adaptive redundancy:

1) *Application critical sections*: In a transaction based workload, if only the transactions update critical data it might make sense to replicate only the transactions in the applications. Similar to traps for virtualizing instructions that can reveal non-native execution, the replica manager can trap on the transaction begin identifier and run the transaction redundantly.
2) *Load on the server*: The workloads can be run in redundant configuration depending on the load on the server. If there are applications with different availability service-level agreements (SLAs), then a priority order can be defined for the applications. If there is a workload surge then only the applications with the most stringent SLAs are run redundantly.
3) *Power and thermal constraints:* The total power consumption or the thermal constraints due to cooling or ambient temperature might determine whether the applications can be run redundantly. For example, when power and thermal constraints would not be violated, then the applications can be run redundantly for better availability.
4) *Error rate*: If a particular core experiences several correctable errors, then applications scheduled on that core can be run redundantly, but when they are migrated to robust cores they need not be run redundantly.

V. RELATED WORK

We discussed most of the related commercial fault tolerant systems and their duplication overheads in Section 2. To the best of our knowledge, this is the first proposal for reducing

the memory duplication in loose lock-stepped processors with I/O level voting. The closest related work to the duplication cache is the memory shadow scheme used by Sequoia fault tolerant systems [14]. Sequoia systems use redundant memory paths and duplicate or shadow all the pages written. Both the processors read from the primary page and the shadow copy is used only for recovery and not for error detection. Furthermore, they duplicate all the pages written by the application all the time, and do not manage the degree of duplication with a duplication cache.

Academic work has proposed several variants of integrated checking at the processor level that is similar to the IBM zSeries approach [15, 17]. Several studies have also evaluated core-level fault detection and containment by running redundant processes, either on a separate core or on a separate thread [8, 9, 18] to reduce the overheads. These approaches typically require custom changes to the processor and do not provide fault tolerance to components other than the core. In contrast, loose lockstepped approaches like ours with I/O level detection are typically less expensive, by virtue of performing fault tolerance at a coarser level. Our work is also different in its approach to leveraging commodity multi-core processors with little additional on-chip support for providing high-levels of availability and fault containment.

With respect to configurable redundancy, to the best of our knowledge, we believe that we are the first to propose selective redundancy by running high and low availability applications simultaneously on a commodity CMP. With regards to adaptive availability, the authors in [15] proposed a technique for opportunistic transient fault detection that runs redundant computations on the same core upon certain microarchitectural events. In contrast, we adapt to system level conditions and guard against error propagation through shared resources.

VI. Conclusions

In this paper we have identified the potential for two techniques to reduce the overhead of redundant execution. For memory, we observed that a large number of pages do not need to be duplicated at all times, and proposed a duplication cache for the current write working set of the application that can reduce overheads significantly. Using detailed simulation results for SPECint and SPECfp benchmarks, we demonstrate the effectiveness of our solution in significantly reducing the overheads from memory, from a 40% reduction in overhead with a performance impact of 4% on average to a 90% reduction in overhead with a performance impact of 5% on average. For cores, using only a portion of the CMP resources for redundant computation and duplication dependent on the varying availability needs of applications for a system operating context can both prove useful. We are currently exploring the implementation of all these ideas in ongoing work.

Acknowledgment

We are grateful to Kyle Nesbit for suggesting the use of a duplication cache and various discussions regarding it. We are thankful to Matteo Monchiero for his invaluable help with the simulator. We would also like to thank Prasun Agarwal, Jung-Ho Ahn, Paolo Faraboschi and Daniel Ortega for their input.

References

[1] M.L., Fair et al., *"Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990"*, IBM Journal of Research and Development, Nov, 2004.

[2] Bernick, D., Bruckert, B., Vigna, P. D., Garcia, D., Jardine, R., Klecka, J., Smullen, J., *"NonStop® Advanced Architecture"*, DSN, 2005.

[3] E. S. Harrison, E. J. Schmitt, "*The structure of System/88, a fault-tolerant computer*", IBM Journal of Research and Development. Volume 26, 1987.

[4] Borkar, S. "*Challenges in Reliable System Design in the Presence of Transistor Variability and Degradation*," IEEE Micro, Nov. 2005.

[5] Ekman, M. and Stenstrom, P., "*A Cost-Effective Main Memory Organization for Future Servers*", IPDPS, 2005.

[6] Gold, B. T. et al., "*The Granularity of Soft-Error Containment in Shared Memory Multiprocessors*", SELSE, 2006.

[7] Meaney, P. J. et al., "*IBM z990 soft error detection and recovery*", IEEE Transactions on Device and Materials Reliability, 2005.

[8] Reinhardt, S. K. et al., "*Transient fault detection via simultaneous multithreading*", ISCA 2000.

[9] Gomaa, M. et al., "*Transient-fault recovery for chip multiprocessors*", ISCA 2003.

[10] Bartlett, W. and Ball, B, "*Tandem's Approach to Fault Tolerance*", Tandem Systems Rev., Feb. 1998.

[11] Smolens, J. C. et al., "*Fingerprinting: Bounding soft-error detection latency and bandwidth*", ASPLOS, 2004.

[12] N. Aggarwal et al., "*Configurable Isolation: Building High Availability Systems with Commodity Multicore Processors*", ISCA, 2007.

[13] N. Aggarwal et al, "*Isolation in Commodity Multicore Processors*," IEEE Computer, June, 2007.

[14] Bernstein, P. A., "*Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing*", Computer 21, Feb. 1988.

[15] Qureshi, M. K. et al., "*Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors*", In Proceedings of the International Symposium on Computer Architecture, (ISCA-32), June, 2005.

[16] Aggarwal, N., Ranganathan, P., Jouppi, N. P., Smith, J. E., Saluja, K. K. and Krejci, G., "*Motivating Commodity Multi-Core Processor Design for System-level Error Protection*", Workshop on Silicon Errors in Logic - System Effects (SELSE), 2007.

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

[18] Vijaykumar, T. N. et al., "*Transient-fault recovery using simultaneous multithreading*", In Proceedings of the 29th International Symposium on Computer Architecture, (ISCA-29), May 2002.

[19] Falcon, A. Faraboschi, P., and Ortega, D., "*Combining Simulation and Virtualization through Dynamic Sampling*", ISPASS-2007.

[20] Purkayastha, A., et al., "*Performance Characteristics of Dual-Processor HPC Systems Based on 64-bit Commodity Processors*", International Conference on Linux Clusters: The HPC Revolution 2004.