Power-Efficient Error Tolerance in Chip Multiprocessors

A redundant multithreading microarchitecture parallelizes the verification process to run on multiple cores while the computation thread runs ahead. This parallelization approach creates significantly less power overhead than other RMT approaches.

M. Wasiur Rashid
Edwin J. Tan
Michael C. Huang
University of Rochester

David H. Albonesi
Cornell University

The microprocessor industry is rapidly moving to the use of multicore chips as general-purpose processors. Whereas the current generation of chip multiprocessors (CMPs)—such as the IBM Power5,[1] Intel's Montecito,[2] and Sun's Niagara[3]—target server applications, future desktop processors will likely have tens of multithreaded cores on a single die. Various redundant multithreading (RMT) approaches exploit the multithreaded capability of current general-purpose microprocessors. These approaches replicate the entire program, running it as a separate thread using time or space redundancy. This guards the processor core against all errors, including those in combinational logic. Because RMT exploits the existing multithreaded hardware, it requires only a modest amount of additional hardware support for comparing results and, depending on the implementation, duplicating inputs.

However, RMT typically involves significant power overhead. The many new RMT designs[4-8] devote little attention to the power dissipation issue, despite power efficiency's criticality in modern processor design. Clearly, with power dissipation and system robustness on an equal footing in future generations, multiprocessors will have to achieve the required level of soft-error tolerance in a more power-aware fashion, while minimally impacting individual program performance. Furthermore, the hardware fabric must be flexible enough for the runtime system to make appropriate decisions on hardware use, given application criticality (in terms of both performance and reliability), as well as the current power envelope, workload mix, and level-of-error vulnerability.

The presence of many cores in current microprocessors also provides opportunities to exploit parallelism. Our approach is a novel twist on the familiar use of parallelism to reduce power through voltage scaling. We base our approach on a CMP microarchitecture (it therefore most closely resembles chip-level redundant threading, or CRT[5]). We divide a computation into a series of chunks and run them on two or more copies of the same hardware to achieve the same processing bandwidth as a sequential execution at a more energy-efficient operating point—for example, through the use of dynamic voltage scaling (DVS). With tens of processor cores on a single die in future microprocessors, harnessing a few to achieve power-efficient error tol-

Published by the IEEE Computer Society

erance will be a viable solution under some runtime conditions. Because each processor core in the CMP microarchitecture is identical, the runtime system can use a given core as a leading compute core, a trailing checker, or in isolation in a nonredundant fashion. Dedicated redundancy solutions such as DIVA[9] don't provide such flexibility.

Implementation challenges

To ensure parallel verification's effectiveness, it's important to let the leading computation thread run far ahead of the verifying threads—in other words, to deeply decouple the computation and verification threads. This creates a large enough verification workload to achieve efficient parallelization. A large slack between the computation and verification wavefronts also prevents any slowdown or overhead in the verification process from "dragging down" the computation. Furthermore, by running far ahead of the verification threads, the lead thread creates a natural prefetch effect for the verification threads because it brings data into the shared L2 cache, therefore making the checker threads' memory latency more fully hidden.

Although this strategy initially appears straightforward, it presents several implementation challenges. Perhaps the largest technical challenge is untangling the complex web of relationships between memory instructions. The many physical threads involved in the process share the same address space and, combined, represent the same semantic thread. However, they operate at different logical times, with the trailing verification threads essentially seeing a past instant of the lead thread and thus seeing the memory state as it was in that past instant (we describe this in more detail later). Thus, out of the same address space, we must provide multiple, consistent memory-state images for the different logical times. (Having the redundant threads each own a distinct address space isn't a viable solution. In addition to wasting precious memory and cache space, this approach would drastically complicate the handling of shared-memory parallel programs because two separate executions of the same parallel program can legitimately generate different memory images. A typical error-detection mechanism would erroneously attribute the differences to soft errors.)

The second major challenge is creating a large slack between the leading and checker threads. This requires a memory-buffering mechanism that holds a large amount of unverified stores and yet supports fast searching and forwarding to ensure correct memory-based dependences. Any added circuitry accessed by more than one core must be latency tolerant to scale well in future technology nodes, and should introduce little additional complexity to the cores themselves. Existing RMT designs use a straightforward fully associative buffer to hold unverified stores.[5,6,8] These designs simply keep the buffer small to permit fast searches for every load. The use of a small buffer doesn't support the deep decoupling required for parallel verification. Yet, simply increasing the buffer size would severely impact load latency and thus performance. Therefore, we need a new solution that permits a large buffer of unverified stores with little impact on the load instructions' critical path.

Architectural support for error tolerance

In our fault-tolerant CMP microarchitecture, every computation thread is executed on a lead processor and a verification copy of the thread is parallelized to execute on multiple checker processors (or *checkers*), which operate at a more energy-efficient point (using lowered supply voltage, increased threshold voltage through body biasing, or both) and reduced frequency. The processors are physically identical but configured slightly differently depending on their role.

As Figure 1 illustrates, we divide the dynamic instruction stream into chunks of consecutive instructions and distribute the chunks to different checkers for parallel verification. To initiate the register state, the lead processor passes checkpoints—that is, lead processor-generated snapshots containing the architectural registers and a few microarchitectural pointers—to the checkers. After the lead processor has executed a certain number of instructions, it freezes the retirement stage and takes a snapshot of the register contents. Given a checkpoint, a checker can start execution as it would if it was loading a new context. The lead processor maintains several checkpoints (⑤ in Figure 2). When it detects a transient error, it can load a checkpoint from before the erring instruction and restart exe-

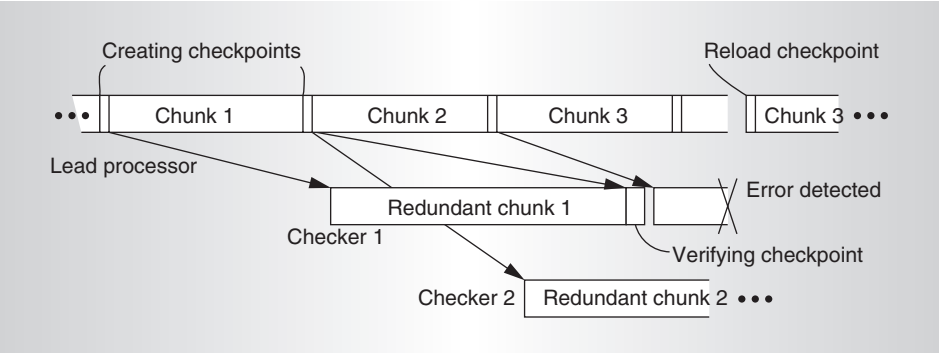

Figure 1. Two-way parallel verification and recovery.

cution. It recycles a checkpoint when there's no possibility of reverting back to that point.

During execution, the checkers re-execute each chunk and verify the lead processor's execution by comparing the address and data of every store in the chunk to those of the original copy executed in the lead processor. When a checker finishes executing the chunk, it compares the register state to that of the next checkpoint. If it finds no discrepancy in the final register state or in the entire sequence of memory updates, it validates the chunk. Otherwise, the checker assumes that a transient error has occurred in the chunk's original or redundant execution. To recover, the processor simply rolls back to the chunk's starting checkpoint. Figure 1 illustrates this process.

Core operation

Figure 2 shows the components of the fault-tolerant CMP.

The lead processor operates much like an ordinary processor—fetching, decoding, scheduling, and executing instructions. However, before it lets the computation's results permanently affect the memory state, the lead processor must verify the results with the checkers' independent computations. Thus,

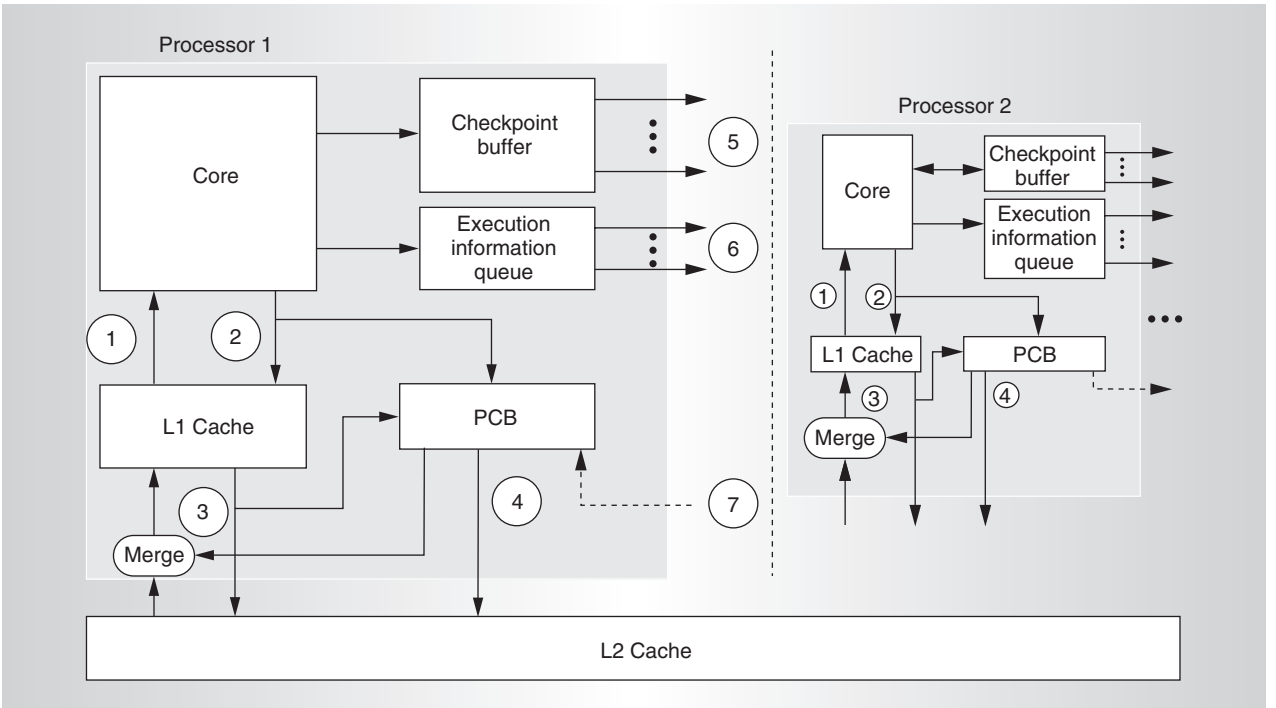

Figure 2. Microarchitecture overview and operation.

when the lead processor commits stores from its store queue (SQ), it places them into the post-commit buffer (PCB) in program order (② in Figure 2). It doesn't release stores until it verifies the entire chunk of instructions. As with the SQ, a core needs to search the PCB to forward results. Thus, the PCB's scalability is a serious issue, and the PCB must be much larger than a typical SQ to provide sufficient decoupling of the lead and checker threads.

To resolve this problem, the cores write to the L1 data cache after stores commit but before verifying the chunk, and at the same time writes them into the slower PCB. Subsequently, when handling a load, the processor only needs to check the SQ and the L1 data cache (① in Figure 2) as in a conventional processor, rather than searching through the larger and slower PCB. Only in the rare case of an L1 miss does the processor probe the PCB (③ in Figure 2) along with the L2 cache. The processor merges the results and writes them into the L1 cache. However, it doesn't let blocks that are written in the L1 data cache but not yet verified propagate to the L2 cache. Thus, when the processor replaces these blocks in the L1 data cache, they are simply discarded. In fact, the very concept of dirty data is irrelevant (in the context of executing sequential code). The PCB performs the "write-back" when the data's correctness is verified (④ in Figure 2). In essence, the store-load forwarding functionality is largely removed from the PCB and assumed by the L1 data cache.

Removing the PCB from the critical load-hit path has two important advantages. It lets us

- implement very large PCBs, enabling significant decoupling without performance loss; and
- construct a single processor and use it in both redundant and nonredundant modes because all of the timing-critical memory operations remain the same regardless of processor mode.

Checkers have the same physical design as a lead processor, run in the same address space, and operate similarly, but differ in three major ways.

- Similar to other RMT designs,[6,7] checkers receive branch outcomes and target addresses from the lead processor through the execution info queue (⑥ in Figure 2). For some applications, this can greatly reduce energy waste for wrong-path instructions. The checker also uses the L1-miss addresses recorded from the lead processor to prefetch at the beginning of the chunk.
- When a store is committed, the checker sends its address and data to the lead processor's PCB for verification. When this verification detects a discrepancy, it triggers a roll-back.
- Finally, when servicing a cache miss, the checker consults the lead processor's PCB instead of its own (⑦ in Figure 2).

PCB operation

The PCB not only enables rapid fault recovery, but more importantly, it's essential to the efficient creation of multiple memory images for the redundant threads. Consider the example in Figure 3. While the lead processor is forging ahead executing chunk 6, the checkers are verifying earlier chunks. At the same moment of wall-clock time, the checkers are actually replaying the lead processor's history, and the shared memory system must provide the memory image as the lead processor saw it when executing the instruction. In other words, when a load executes in a checker, the SQ, the PCB, and the memory hierarchy should together capture the effect of all stores prior to the load and none thereafter. With the help of the PCB, we only store the images' common part in the L2 cache and the rest of the memory subsystem, and construct the most up-to-date image on-the-fly based on the processor initiating the request. The processor's identity determines which logical time pointer of "now" to use.

To enhance efficiency, all cores use their private L1 cache to store data consistent with their view of the memory image. This lets them access the PCB only when a miss occurs. Cache lines must therefore contain the most up-to-date data from that processor's viewpoint. So, when a checker needs to fill a cache line, it obtains the data from the L2 cache and searches the PCB for appropriate updates. This is called a *backward search* (see Figure 3). To get an up-to-date cache line, the checker must obtain the most recent writes to all

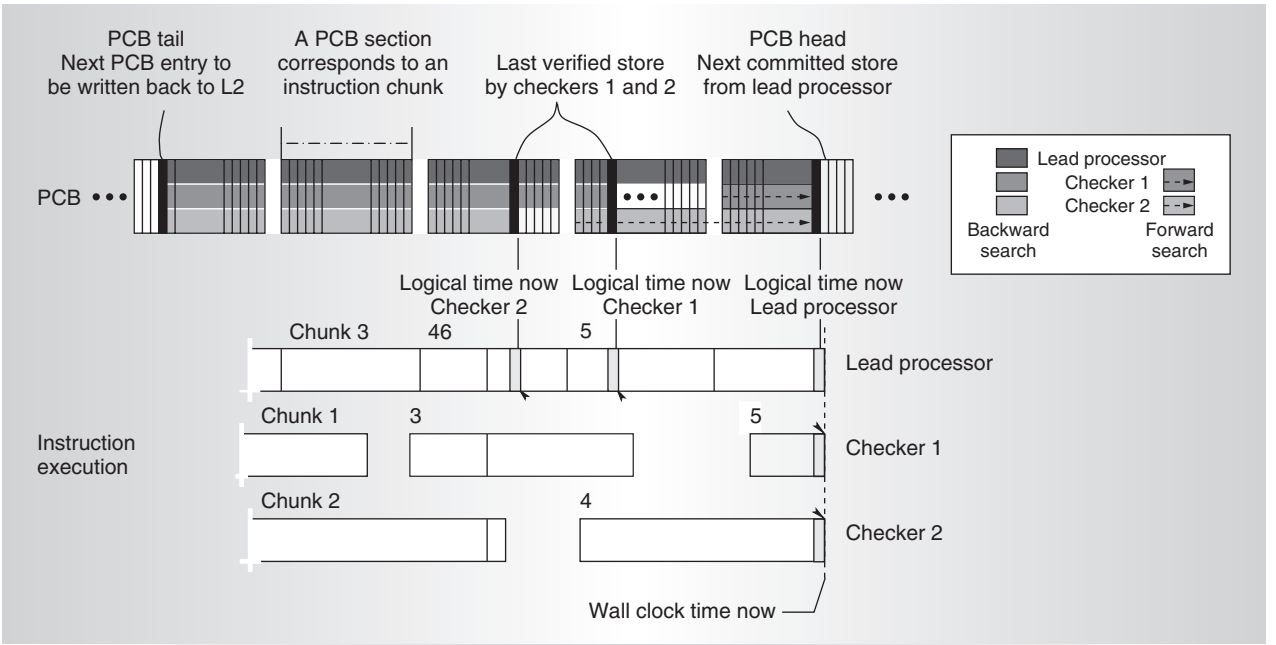

Figure 3. Logical time frames from the different cores' perspectives. At any moment, a checker's logical time is in the lead processor's past. Correspondingly, the checkers only see part of the memory state in the post-commit buffer (PCB, shown in different shades) in the backward search. A forward search starts from the next section of the PCB (corresponding to the next chunk of instructions).

words in the line in the backward search.

To avoid the circuit complexity of returning multiple words with different offsets from different PCB entries, we use a PCB that can only return one word at a time. When we search the PCB for the word being loaded, we can simultaneously detect whether the PCB contains other words of the same line. If so, we increment the word address (with wraparound) and probe the PCB multiple times until we cover the entire line. This, of course, delays the cache line's availability except for the critical word. Without multiple probes, we can't cache the line. This would lead to pathologically high performance degradation and thus isn't a viable solution. If the cache has per-word valid bit support, however, we can avoid multiple PCB probes by caching only the word being loaded. Our simulation shows that this only marginally degrades performance and energy metrics.[10]

Parallelizing the verification thread onto multiple checkers creates a peculiar cache-coherence issue. When a checker is verifying a chunk of instructions, other checkers verifying the same program will not execute the stores in that chunk, resulting in some data in the L1 data cache becoming stale when the checkers execute subsequent chunks. To solve this problem, we can leverage the built-in cache-coherence protocol in a chip-multiprocessor. The lead processor sends a quasi-invalidation message the first time every cache line is written to in a chunk. When a checker receives the invalidation, it sets a volatile bit if it finds the cache line in its L1 cache. This indicates that the cache line is being updated in a future instruction chunk. Therefore, when the checker skips one or more chunks to verify another chunk, all volatile lines are invalidated because they might be modified in the skipped chunks. Also, when filling a cache line, a checker sets the volatile bit when the search in PCB reveals that the line will be modified in a future chunk. This requires a *forward search* in the PCB (Figure 3). The forward search requires little additional support. Finally, we can optimize this basic protocol to reduce unnecessary invalidations.[10]

Because the PCB is large and accessed in an associative manner, each access takes significant time and energy; thus, reducing unnecessary PCB accesses is desirable. Our empirical findings show that many PCB searches return no

Table 1. Simulation parameters.

Feature Size	
Fetch queue size	16 instructions
Fetch/dispatch/commit	4/4/12
Combined branch predictor	4,096-entry bimodal, 2-level adaptive (1,024 L1, 4,096 L2), and 4,096-entry meta predictor
Branch target buffer/return address stack	2,048 entries two-way /32 entries
Branch misprediction latency	15 cycles
Integer units	3 arithmetic logic units (ALU), 1 multiplication unit
Floating-point units	3 ALU, 1 multiplication unit
Register file	128 integer, 128 floating-point
Issue queue	32 integer entries, 32 floating-point entries
Load-store queue (LSQ)/Re-order buffer (ROB)	64 entries/256 entries
L1 I/D cache	32 Kbytes, two way, 32-byte line, 2 cycles
L2 cache (shared)	8 Mbytes, 32 way, 128-byte line, 20 cycles
TLB (I/D each)	128 entries, 8 Kbytes, fully associative
Memory latency	250 cycles
Post-commit buffer	128 entries per chunk, 8 chunks, 1 port, 8 cycles per search
Chunk size	2048 instructions or 128 stores, whichever comes first
Membership hash table	257 entries, 8 bits per entry
Checkpoint creation/loading	16 cycles (4 registers per cycle)

hit. This is especially so for the searches initiated by the lead processor: on average, no more than 0.3 percent of these searches return a match. A quick membership test using a hash table can filter out unnecessary PCB searches.[10]

Finally, in its current form, our design only supports redundant execution of applications with a single semantic thread. We will extend the support to parallel applications in the future.

Experimental setup and analysis

To evaluate the proposed architectural support for fault tolerance, we simulate a CMP in which one core runs as the lead processor with various configurations of checker processors. We use a modified version of the SimpleScalar 3.0b toolset simulating the Alpha AXP ISA with structures for intercore communication. Table 1 lists the parameters for a single CMP core. We interconnect the cores using a ring with a bandwidth of 128 bits per cycle. We performed the experiments using the Standard Performance Evaluation Corporation (SPEC) CPU2000 benchmark suite.

To evaluate energy consumption, we model both dynamic and leakage energy in detail. We use Wattch[11] to estimate the dynamic energy component, whereas leakage energy is temperature dependent and based on predictive SPICE circuit simulations for 45-nanometer technology using BSIM3.[12] We base device parameters, such as V_{th}, on 2003 International Technology Roadmap for Semiconductors projections for the year 2010. ITRS forecasts a 45-nm technology node with a 15-GHz clock frequency. We model temperature (for leakage calculations) using HotSpot[13] with the Compaq Alpha 21364 as the core floor plan.

Except in special-purpose environments, fault-tolerant systems spend most of the time in fault-free situations. For example, at sea level, cosmic particle flux is on the order of 10—100 n/cm^2 hour.[14] Even pessimistic estimation would translate this to no more than a few potential upsets per minute. In terms of microprocessor clock cycles, this constitutes an exceedingly rare event. Therefore, our experimental analysis focuses on the design's performance and energy characteristics in fault-free situations. We show that even when soft errors occur as frequently as once per millisecond, the rollback overhead is still negligible.[10]

The energy-efficient fault-tolerant configuration consists of a lead processor and two checkers. The checkers run at half frequency to verify the computation in parallel. In this study, the supply voltage is appropriately

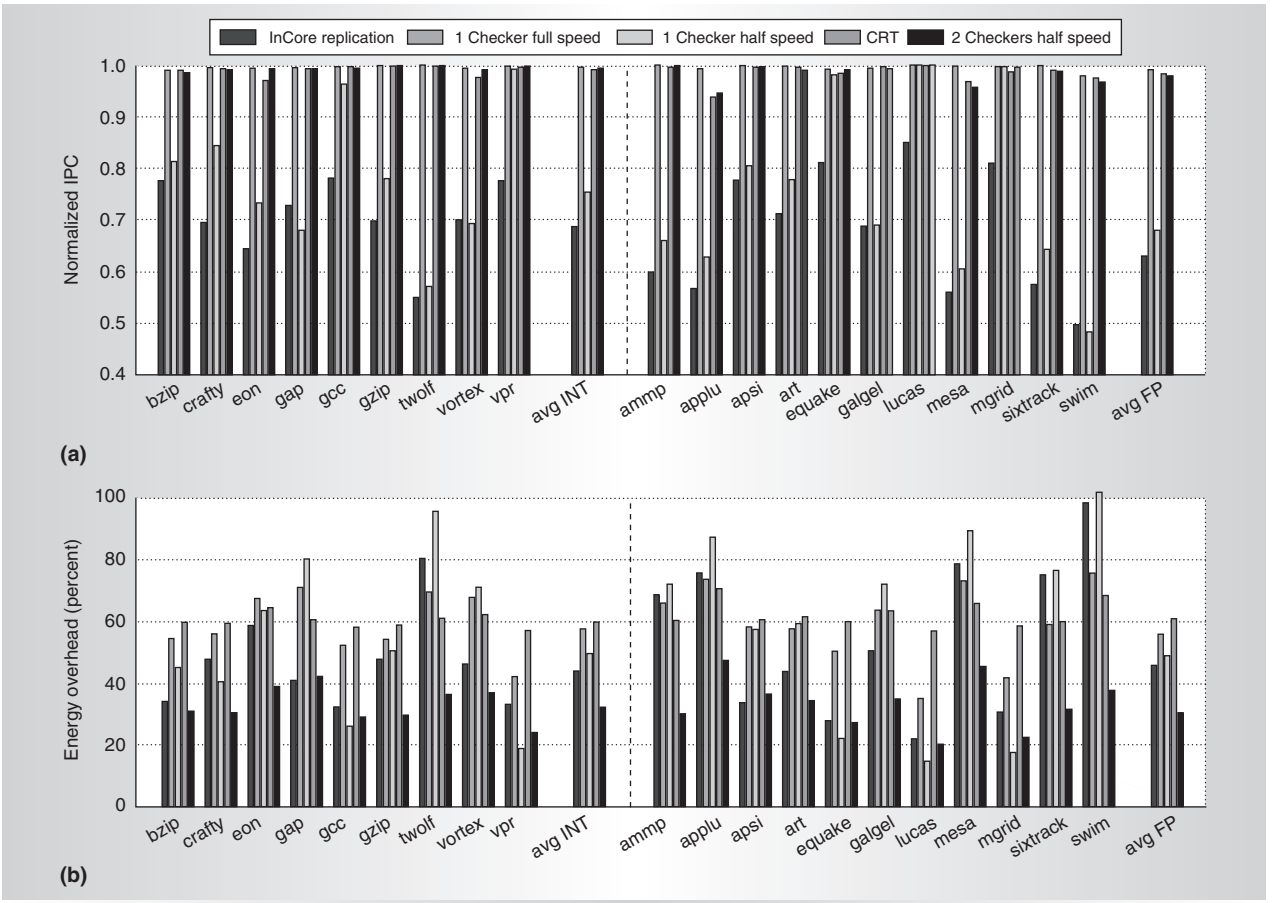

Figure 4. Effect of using different fault-tolerant configurations on various applications' performance (a) and energy (b).

reduced with the frequency. We compare this configuration with several others: a checker at full frequency; a checker at half frequency; a design modeled after CRT[5]; and, as a reference, an instruction-level in-core replication mechanism[15] offering more limited protection. In all cases, the lead processor runs at full frequency. Results are normalized to that of a single core running at full frequency without any fault-tolerance mechanism.

Figure 4 shows the performance and energy overhead with fault-tolerant operation. We can make three observations from this figure.

First, the performance degradation of using two half-speed checkers is imperceptible: the IPC (instruction per cycle) of redundant execution normalized to that of nonredundant execution is close to 1 for all applications. This is because the overhead of checkpoint creation is negligible, and the checkers provide sufficient verification bandwidth and thus don't slow the lead processor. Furthermore, if the verification mechanism can't keep up (such as when running a single half-speed checker), not only does performance degrade, but the energy overhead increases because the power-hungry full-speed lead processor runs longer and burns more power in clock distribution and leakage.

For example, with *swim*, using a single half-speed checker slows the lead processor so much that the energy overhead is about three times that of using two checkers. On the other hand, for applications such as *lucas* and *mgrid*, the lead processor's prefetching effect makes one half-speed checker fast enough that the overall performance of using a single half-speed checker isn't much different from that of using two. In such cases, the extra checker doesn't save much energy, if any.

Second, parallelizing the checking process significantly improves the energy efficiency of fault-tolerant operation. Compared to a roughly 60-percent energy overhead using one

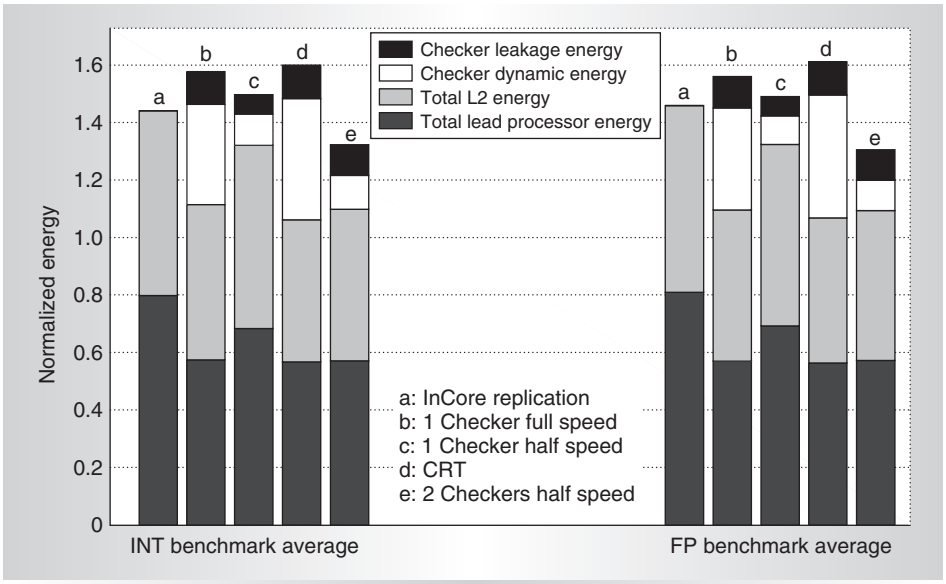

Figure 5. Normalized energy consumption by category.

checker at full speed, the parallel verification approach incurs only about 31-percent energy overhead. (Discounting the energy contribution from the shared L2 cache, these overheads become 96 and 45 percent, respectively.) In fact, the energy overhead of using two half-speed checkers is even lower than the overhead of the in-core replication technique.[15]

Third, even with one full-speed checker, our approach achieves noticeably lower energy overhead than a CRT-like system. This is mainly because of the PCB's deep decoupling capability: Even when the checker isn't working, the lead processor can execute tens of thousands of instructions without stalling. This gives us the flexibility to orchestrate instruction execution on the checkers to improve energy efficiency without impacting the lead processor's performance. For example, when the lead processor's PCB is close to empty, we can set the checker core to sleep mode or use it for other tasks altogether until the PCB is close to full again. Because of assisted execution, the checker can drain the PCB faster than the lead processor fills it. Indeed, we found that a full-speed checker is idle on average 32 percent of the time and as much as 80 percent of the time. Once idle, the checker can stay idle for an average of 4,300 cycles at a time. To model the effect of sleep mode or the use of the otherwise idle checker for other tasks, we assume that the checker consumes no dynamic energy in these long idle periods. Without a large PCB in a CRT-like design, the trailing thread is much more tightly coupled to the leading thread and doesn't have these long idling periods, making it impractical to apply sleep states or use it for other tasks.

To better illustrate the energy overhead reduction, Figure 5 further breaks down the normalized energy consumption into different components: the lead processor's energy consumption, the L2 cache's energy consumption, the checkers' leakage energy, and the checkers' dynamic energy consumption. For clarity, we show only the averages of the integer and floating-point applications. The variation from application to application is moderate. Clearly, using two voltage-scaled, half-speed checkers consumes much less dynamic energy—about 34 percent that of one full-speed checker. The reduction in supply voltage (from 1 to 0.6 volt) is the main factor in this reduction. Thanks to the "perfect" branch prediction and lead processor prefetching, the checkers also execute fewer instructions and have shorter run. These factors also help reduce energy consumption.

The reduced supply voltage combined with lower temperature due to much lower power consumption significantly reduces the per-core leakage power. However, the combined leakage of two cores stays largely the same as

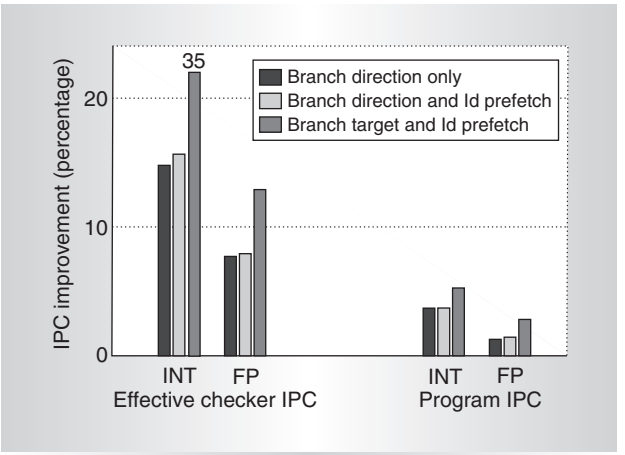

Figure 6. Performance impact of assisted execution.

that of a single full-speed core. The overall result is that the energy cost is lower than that incurred by techniques with more limited replication (for example, the in-core replication scheme doesn't replicate instruction fetch and decode[15]). This is because even though the redundancy is more limited, it's provided on more power-hungry hardware. Furthermore, in the in-core replication design, competition for the same resources increases execution time, which in turn increases per-cycle energy costs due to leakage, clock distribution, and so on.

Evaluating design options

Several design options are possible in terms of architectural support. As in the previous section, we compare these options in terms of average results of the integer and floating-point applications.

Effectiveness of assisted execution

The lead processor passes various information, including branch outcomes, branch target addresses, and L1 data cache miss addresses, for load prefetch. Figure 6 shows the performance impact of different forms of assistance. We compute the checker's effective IPC by excluding idle periods with no verification workload. This is a useful metric because although the checkers aren't a bottleneck in the experiments, in a real-world scenario, the system might multiplex different verification workloads onto the same set of checkers, in which case checker performance can be important. We normalize the results to a system in which none of this information passes from the lead processor to the checker.

With assisted execution, the checkers are up to 35-percent faster on average. Information about branch outcomes is indeed useful. Its availability improves the effective IPC by about 10 percent. L1 prefetch information isn't as useful, improving only about 1 percent of the effective IPC, although it requires little bandwidth to communicate that information. Branch-destination information is helpful, especially for integer applications, improving checker performance by another 20 percent. However, communicating that information can require a lot of bandwidth as well as energy. Finally, the impact of assisted execution on overall program IPC is small—from 1 to 5 percent. This indicates that indeed the two checkers aren't the bottleneck in this configuration and thus have processing power to spare.

Further reduction of complexity

Intuitively, because the leading computation passes on branch information and helps prefetch data into caches, it's much easier for the checkers to exploit instruction-level parallelism (ILP). Therefore, we might be able to reduce checker design complexity by disabling some of the core ILP mechanisms. This would reduce checker energy consumption without much performance impact on the lead processor.

We evaluate two possibilities:

- We disable load speculation and dynamic disambiguation and force memory instructions to execute in order.
- We turn off out-of-order scheduling (issuing instructions in order).

Figure 7a shows the degradation in the effective IPC of the checkers and in the entire program IPC under these simplifications. Figure 7b shows the increase in energy consumption. In all cases, the baseline is the default configuration without any complexity reduction measure. Surprisingly, these simplifications result in no net energy gain. In fact, the energy increases. There are two reasons for this:

- The simplifications slow both the checkers and the lead processor. The added execution time increases energy overhead, especially for the lead processor.

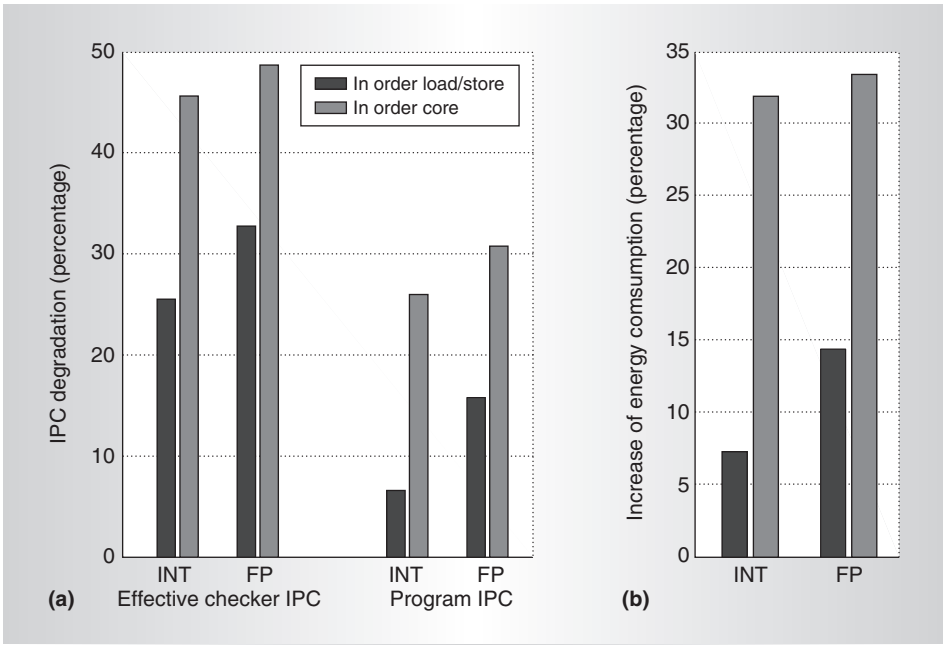

Figure 7. Impact on performance (a) and energy consumption (b) of disabling various instruction-level parallelism (ILP) mechanisms in the checker cores .

- Because of the voltage scaling of the slower checkers, the checkers yield little additional energy savings.

Thus, we can achieve most of the energy savings by simply using the per-processor DVS mechanisms already being incorporated into next-generation multicore chips,[2] without resorting to dynamic simplification of the checker cores.

In the near future, system reliability will quickly become a major design concern and has to be addressed at both the circuit and architecture levels as a first-class design constraint. Redundant Multi-Threading effectively exploits multi-threaded microprocessors for both fault detection and recovery, yet prior approaches are either inflexible or energy-inefficient. As power consumption is already the limiting factor in high-end processors, energy-efficient fault tolerance can no longer be regarded as an oxymoron, but as a challenge that has to be addressed. The forthcoming integration of tens of processor cores on a single die presents an opportunity to exploit the inherent parallelism in correctness verification. We have described a flexible platform using novel mechanisms and modest hardware support to enable on-demand, energy-efficient redundant execution. In future work, we plan to extend the support to parallel applications. MICRO

References

1. R. Kalla, B. Sinharoy, and J. Tendler, "Simultaneous Multithreading Implementation in Power5—IBM's Next-Generation Power Microprocessor," *Proc. Hot Chips 15*, IEEE CS Press, 2003, pp. 293-303.
2. C. McNairy and R. Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," *IEEE Micro*, vol. 25, no. 2, Mar./Apr. 2005, pp. 10-20.
3. P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, Mar./Apr. 2005, pp. 21-29.
4. M. Gomaa et al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 98-109.
5. S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2002, pp. 99-110.
6. S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multi-

threading," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2000, pp. 25-36.
7. E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, 1999, pp. 84-91.
8. T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery via Simultaneous Multithreading," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2002, pp. 87-98.
9. T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 1999, pp. 196-207.
10. M. Rashid et al., "Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2005, pp. 315-325.
11. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2000, pp. 83-94.
12. BSIM Design Group, *BSIM3v3.3 MOSFET Model—Users' Manual*, Jul. 2005, http://www-device.eecs.berkeley.edu/~bsim3/ftpv330/Mod_doc/b3v33manu.tar.
13. K. Skadron et al., "Temperature-Aware Microarchitecture," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 2-13.
14. J. Ziegler et al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)," *IBM J. Research and Development*, vol. 40, no. 1, Jan. 1996, pp. 3-18.
15. J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 2001, pp. 214-224.

M. Wasiur Rashid is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Rochester. His research interests include energy-aware and fault-tolerant microprocessor architectures. He has an MS in Electrical Engineering from the University of Rochester. He is a student member of IEEE. Contact him at rashid@ece.rochester.edu.

Edwin J. Tan is a doctoral candidate in the Department of Electrical and Computer Engineering at the University of Rochester. His research interests include fault-tolerant computer architectures, digital and analog VLSI circuit design, and image processing. He has an MS in Electrical Engineering from the University of Rochester. He is a student member of IEEE, TBP, and HKN. Contact him at etan@ece.rochester.edu.

Michael C. Huang is an assistant professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Rochester. His research interests include computer system architecture and processor microarchitecture, with emphases on adaptive architecture, power-aware design, reliability, and system optimization. Huang has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society and the ACM.

David H. Albonesi is an associate professor in the Computer Systems Laboratory at Cornell University. His research interests span single and multiple processors and threads, with particular interests in adaptive, power-efficient, and reliability-aware computer architectures, and microarchitectures exploiting new technologies. Albonesi has a PhD in computer engineering from the University of Massachusetts Amherst. He is a senior member of the IEEE Computer Society and a member of the ACM.

Direct questions and comments about this article to Michael C. Huang, 414 Computer Studies Building, Department of Electrical and Computer Engineering, University of Rochester, Rochester, NY 14627; michael.huang@rochester.edu