Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance

M. Wasiur Rashid, Edwin J. Tan, and Michael C. Huang
Department of Electrical & Computer Engineering
University of Rochester
{rashid, etan, michael.huang}@ece.rochester.edu

David H. Albonesi
Computer Systems Laboratory
Cornell University
albonesi@csl.cornell.edu

Abstract

As device dimensions continue to be aggressively scaled, microprocessors are becoming increasingly vulnerable to the impact of undesired energy, such as that of a cosmic particle strike, which can cause transient errors. To prevent operational failure due to these errors, system-level techniques such as redundant execution will be increasingly required for fault detection and tolerance in future processors. However, the need for redundancy is directly opposed to the growing need for more power efficient operation. Conventional techniques that use multi-core microarchitectures to provide whole-thread duplication generally incur significant energy overhead which can exacerbate the already severe problem of power consumption and heat dissipation given a certain throughput requirement. In the future, approaches that supply the necessary level of robustness at a given throughput level must also be power-aware.

We propose a thread-level redundant execution microarchitecture that significantly reduces the energy overhead of replication without unduly impacting performance. Our approach exploits the fact that with appropriate hardware support, the verification operation can be parallelized and run on a chip multiprocessor with support for frequency scaling together with supply voltage scaling and/or body biasing. To further improve the efficiency of verification, we exploit the information obtained by the leading thread to assist the trailing verification threads. We discuss in detail the required architectural support and show that our approach can be highly energy-efficient: using two checkers, fully replicated execution costs only an average 28% extra energy over non-redundant execution with virtually no performance loss.

1 Introduction

Each new process generation provides increased transistor density but with a reduction in the critical charge required to maintain proper device state. As a result, microprocessors are becoming increasingly vulnerable to disturbances from unwanted energy from the environment. In recent years, significant concern has arisen over particle strikes (*e.g.*, from neutrons and alpha particles) that may result in *transient* or *soft* errors. Although such problems have been dealt with for decades in machines used in particularly vulnerable environments (such as in deep space), the usual solutions of shielding, radiation hardening, or other advanced manufacturing techniques are very costly for general-purpose systems. Redundancy at the architectural level, used today in specialized database servers, will become more prevalent in future general-purpose microprocessors as a means to detect, and recover from, particle disturbances.

The majority of disturbances in today's general-purpose microprocessors occur in memory elements such as caches and main memory DRAMs. Error detection and correction codes are commonly implemented to prevent failure in the event that a particle strike changes the internal memory state. However, recent studies indicate that the rate of potential soft errors in logic components will rapidly increase in future technology generations [20], which will call for more global solutions to the microprocessor soft error problem. Unlike memories where error codes can be implemented at a reasonably low cost, the usual solution for logic is to employ some sort of replication. In the various *thread-level redundancy* solutions that have been proposed in recent years (*e.g.*, [8, 16, 17], Simultaneous Multi-Threading (SMT) or Chip Multi-Processing (CMP) is used to provide coarse-grain instruction replication in a fairly straightforward manner, with a modest amount of additional hardware support. In these approaches, the program is replicated and the results (or a subset) from the multiple program copies are compared to detect errors, and in some approaches, to correct them. The main advantage of thread-level redundancy is that it exploits the natural time or space redundancy inherent in an SMT or CMP implementation. However, these approaches are largely "power agnostic" in that, given a particular throughput requirement, replicating the program and running it on identical hardware significantly increases the power consumption required to complete a given set of tasks. Clearly, with power dissipation and system robustness on an equal footing in future microprocessor generations, new approaches must be devised that achieve the required level of soft error tolerance in a more power-aware fashion, yet with very little impact on individual program performance.

In this paper, we propose an approach that leverages the redundancy available in a CMP in a more power-efficient manner than prior approaches. The key observation is that given appropriate hardware support, the task of verifying correct computation can be parallelized and run on multiple processing cores set at a slower, but more energy-efficient, operating point. CMPs are expected to have tens of cores in the future, and thus harnessing a small subset to provide error tolerance when demanded by the application may be a reasonable choice given a significant energy efficiency advantage. Parallelization of the verification process also

permits operation at virtually an identical level of performance as non-redundant computation, in contrast to prior approaches that suffer non-trivial performance losses.

A key enabling factor to the effectiveness of our approach is permitting the leading computation thread to run far ahead of the verifying ones. This is necessary to create a large enough "verification workload" that can be efficiently divided up and run in parallel on multiple cores, and to allow the memory latency to be fully hidden by the prefetching generated by the lead thread. The key challenge in creating such a large slack lies in the design of an efficient memory buffering mechanism that holds a large amount of unverified stores, and yet supports fast searching and forwarding to ensure correct memory-based dependences. We propose a novel memory buffering and disambiguation scheme that uses the L1 data cache in a speculative fashion to hold unverified stores. This removes the associative searching necessary in the disambiguation process from the critical path, which in turn significantly decouples the execution of the leading and the trailing threads. Finally, we exploit the ability of the leading thread to provide information to speed up the execution of the trailing verification thread. Overall, these various mechanisms permit fully replicated operation with virtually zero slowdown and only an additional 28% extra energy overhead compared to a processor with no redundancy.

The rest of the paper is organized as follows. Section 2 discusses our approach at a high level and provides an overview of the system, while Section 3 describes our proposed design in detail. Section 4 presents the experimental methodology, while our results are presented in Section 5. Related work is presented in Section 6, and we conclude in Section 7.

2 Providing Flexible and Efficient Thread-Level Redundancy

In thread-level redundancy (TLR), the entire program thread is replicated and run under time or space redundancy. TLR provides protection to the entire processor core, whereas other more local techniques, such as the use of checksums with functional units, do not protect control paths. Given that a large portion of the core is devoted to control logic, protecting just the datapath or individual local functions is insufficient. TLR is also compatible with the multi-threaded approach, in the form of Chip Multi-Processing (CMP), Simultaneous Multi-Threading (SMT), or a combination of both, that is rapidly becoming dominant in microprocessors.

Nevertheless, the power costs of TLR are a major concern. Replicating the entire thread (perhaps more than once, as in triple-modular redundancy), comes at the high cost of increased power dissipation at a time when power is perhaps *the* limiting factor in achieving greater microprocessor performance levels.

Therefore, future TLR designs must be power-aware. Inflexible designs such as lock-stepping are inherently power inefficient. A more decoupled approach which allows one thread to lead and pass on information to optimize the trailing thread can greatly enhance power efficiency. Passing branch outcomes is already used in some TLR designs (*e.g.*, [11, 17]). Other examples include the use of the results of long-latency instructions or scheduling hints. We also envision a flexible design where an overall control mechanism can judiciously determine the optimal tradeoff between power, performance, and dependability at runtime. As one example, otherwise idle cores can be harnessed to perform verification of the lead thread in parallel using a lower speed, more energy-efficient, operating point.

Our approach to power-aware, thread-level fault tolerance requires minor modifications to the processor core pipeline, and consists of two major components: *parallel verification* and *assisted execution*.

We exploit a widely-used strategy in energy-efficient computing: parallel processing. When processing is parallelized, the same processing bandwidth can be achieved at a slower, more energy-efficient operating point, for example through the use of Dynamic Voltage Scaling (DVS). We exploit the fact that verification is inherently deeply parallel, and divide the sequential instruction stream into a series of chunks to be verified in parallel. Once all chunks are verified, the correctness of the entire stream is instantly established. In practice, however, there are several challenges to implementing efficient parallel verification. Perhaps the largest challenge is buffering and searching unchecked memory stores: the performance requirement of a small, fast, buffer limits the amount of unchecked instructions that can be parallelized. We present a novel approach that uses the L1 data cache to facilitate disambiguation. This approach removes the buffer from the speed-critical L1 hit path, thereby permitting a large buffer to be used (to create a large "verification workload") with virtually no performance loss.

Another strategy for optimizing the verification process is assisted execution, in which information acquired in some form of pre-execution permits more efficient operation. Examples of assisted execution include decoupled architectures [23], slipstream [26], and various "helper thread" implementations (*e.g.*, [2, 6]). In a decoupled TLR design, assisted execution is straightforward: one thread runs ahead and executes the entire program and provides information to the replicated trailing thread. Barring transient errors, this information is always accurate. However, in a realistic design, the key issue is the tradeoff between the benefit of the information and the cost to gather it and communicate it. In our design, we pass on branch information and L1 cache prefetch hints.

3 Architectural Support

3.1 Verification and Recovery Overview

The architectural support for power-efficient redundant multithreading builds on a standard CMP with homogeneous processor cores. Every computation thread is executed on a *lead processor* and a verification copy of the thread is executed on one or more *checker processors* (or checkers for short). The additional support for redundant threading is largely off the critical timing path and built into every core, permitting a single core with the same physical design to serve in the role of lead processor, checker, or by itself in a non-redundant mode.

Since verification threads incur additional energy over non-redundant execution, it is desirable to operate the checkers at a more energy-efficient operating point to reduce this energy cost. To achieve this, we use a lower supply voltage and/or increased

threshold voltage (through body biasing) for the checkers. Consequently, their operating frequency must be reduced. In order for the checkers to keep up with the lead processor, we parallelize the workload by dividing the dynamic instruction stream into *chunks* of consecutive instructions, and distribute different chunks to multiple checkers for parallel verification. This is done by passing checkpoints – snapshots of architectural state generated by the lead processor – to the checkers. Given a checkpoint, a checker can start execution in the same manner as loading a new context after a context switch.

During execution, the checkers re-execute each chunk and verify the lead processor's execution by comparing the address and data of its stores to those of the lead processor. These stores are buffered in the lead processor's *Post-Commit Buffer* (PCB) before the entire chunk is verified to be correct. Once the checker executes its chunk in its entirety, checker compares its architectural state to the next checkpoint generated by the lead processor. If a discrepancy is found in the comparison of stores or between the checkpoints, then a transient error is assumed to have occurred in either the original or the redundant execution of the chunk. To recover, we simply roll back to the starting checkpoint of the chunk. This process is illustrated in Figure 1.

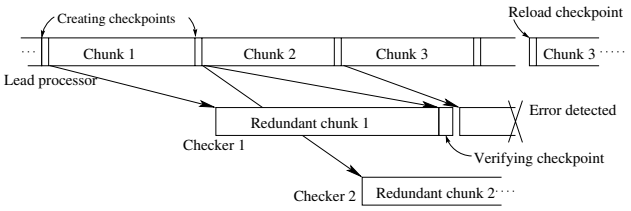

Figure 1. An example of two-way parallel verification and recovery. The lead and checker threads are delayed by a chunk as explained in Section 3.3.

In our system, a checkpoint only contains the content of the architectural registers and a few microarchitectural pointers and serves three purposes: to kick off redundant execution in a checker, to detect errors, and to safe-keep state for roll-back recovery. As with other roll-back schemes, our mechanism can detect and recover from infrequent transient errors. If there are hardware defects or the error rate is extremely high, such an implementation may not guarantee forward progress. Special support is needed to handle these cases.

In the next three subsections, we discuss the operation of the lead processor, the checker, and the PCB. The added logic for fault tolerance is shown in Figure 2. The circled numbers in this figure correspond to those in the text that follows.

3.2 Operation of the Lead Processor

The lead processor operates very much like an ordinary processor: fetching, decoding, scheduling, and executing instructions. However, before we allow the results of the computation to permanently affect the memory state, they need to be verified with the independent computation performed by the checkers. Thus, when stores are committed from the Store Queue (SQ) of the lead processor, they are placed into the PCB in program order (②).

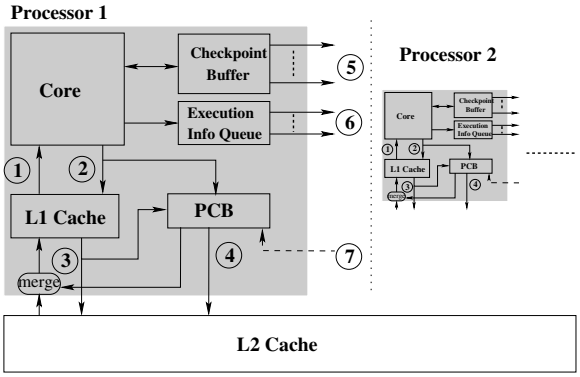

Figure 2. Diagram showing the architectural support for redundant threading.

Stores are not released until the entire chunk of instructions are verified. As with the SQ, the PCB needs to be searched in order to forward results as needed.

A buffer similar to the PCB is necessary whenever the lead and checker threads do not retire stores in lock step but are permitted to slip with respect to one another. Such a buffer is used in various previously proposed fault-tolerant designs (*e.g.*, [8]). However, what is not addressed in these prior designs is the scalability issue of the buffer. The PCB shares the serious issue of scalability with its pre-commit counterpart, the SQ [19]. Every load is required to check the SQ for memory disambiguation. If the queue is too large, the associative search may take longer than accessing the L1 data cache and thereby degrade performance. The addition of the PCB further exacerbates this problem as it also needs to be checked and its datapath merged with those of the cache and SQ. Furthermore, with more than one checker operating in parallel, the PCB needs to be large enough to permit the lead processor to run far ahead to create a large enough verification workload for the checkers. The delay in accessing such a large, associative PCB can easily exceed that of an L1 data cache access.

To address this issue, we employ the L1 data cache to perform most forwarding tasks as follows. When stores are committed, in addition to placing them in the PCB, we allow them to update the L1 data cache (②). This permits subsequent loads of the same data to be obtained through checking the SQ and the L1 data cache for matching addresses (①) just as in a conventional processor, rather than searching through the larger and slower PCB. Only in the infrequent case of an L1 miss is the PCB probed (③) along with accessing the L2 cache. The results of these parallel accesses are merged and written into the L1 cache (more details in Section 3.4). In this sense, the PCB serves as a victim buffer. Since stores written in the L1 data cache are yet to be verified, they are not allowed to propagate to the L2 cache. Thus, when a dirty cache line is replaced in the L1 data cache, it is not written back to the L2 cache. Instead, it is simply discarded. In fact, in this context, the very concept of dirty data is irrelevant since the "write-back" is performed by the PCB after the correctness of the data is verified (④). 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 access path has two main advantages. First, it permits very large PCBs to be implemented without undue performance loss, and thus for the lead processor to run far ahead of the checkers. Large verification workloads can thus be created, parallelized, and run in a low power mode on several checkers if desired. Second, it permits a single processor to be constructed and used in non-redundant mode, in redundant mode as the lead, and in redundant mode as the checker, which simplifies the overall CMP design.

Each processor is equipped with a checkpoint buffer that maintains checkpoint state. Each entry of the buffer is a storage array that captures the contents of the architectural registers including the program counter. The lead processor periodically creates a checkpoint by freezing the commit stage of the processor and copying the architectural registers to the checkpoint buffer entry. In microarchitectures with a dedicated architectural register file (*e.g.*, Pentium III), this is straightforward. Otherwise, a retirement map [9] is required, but this need not be updated every cycle. Rather, it can be constructed on demand by retrieving the checkpoint of the front-end map corresponding to the branch closest to retirement, walking toward the head of the reorder buffer (containing the oldest instruction), and accordingly updating the mapping. Of course, given the opportunity, we can create a checkpoint right before a branch where we can get a ready-made map copy without extra updates.

The entries in the checkpoint buffer are allocated and recycled in FIFO order. When the execution up until the point of checkpoint i has been verified, checkpoint $i-1$ is no longer needed and can be recycled. When a checker notifies the lead processor of an error in a particular chunk, the lead processor aborts the current execution and rolls back to the prior checkpoint, and instructs all associated checkers to abort as well. This rollback operation includes flushing the pipeline and invalidating the L1 data cache for the lead processor and all checkers. This invalidation is necessary because the L1 data cache of the lead processor and some of the checkers contain future (and thus incorrect) data with respect to the point of roll back. Performing a full cache invalidation also simplifies corner-case handling for cache coherence (Section 3.4). After the pipeline flush and cache invalidation, we restore the appropriate checkpoint and restart execution. Additional microarchitectural information, such as the write pointer to the PCB which points to the entry for the next committed store, is also restored from the checkpoint. As we demonstrate in Section 5, assuming that errors are relatively infrequent, full cache invalidation has little overall performance impact.

As the lead processor usually runs far ahead of the checkers, it can serve as a helper thread to assist the checkers in executing more efficiently, for example, by reducing their branch mispredictions and cache misses. One natural effect of the deep decoupling between the threads is that the lead thread creates prefetches into the shared L2 cache for the checkers. Therefore, in terms of reducing cache misses, we only focus on L1 misses. We record the lead processor miss addresses for every chunk of instructions in the *Execution Info Queue* (EIQ). When the starting checkpoint is dispatched to a checker, it includes this miss address array so that the checker can prefetch from these addresses. For branches, similar to prior approaches [16, 17], we communicate branch outcomes from the lead processor to the checkers also via the EIQ (⑥).

3.3 Operation of the Checkers

As noted earlier, the lead and checker cores share the same physical design, and operate slightly differently. When a core is used as a checker, it performs redundant execution from checkpoints generated by the lead processor. It runs in the same address space as the lead processor, and thus the TLB and other aspects of address translation are handled much like running a parallel program. The content of the registers is copied from the checkpoint using the same functionality that handles a rollback. The only difference is that the checkpoint is fetched from a different processor's (the lead processor's) checkpoint buffer (⑤).

Starting from the checkpoint, the checker fetches and executes instructions as in a conventional processor. The branch predictor, however, is disabled and branch predictions are provided by the lead processor through the EIQ (⑥). The branch destination addresses could be supplied from the buffer as well, but would require much inter-processor communication bandwidth and a larger branch buffer. Thus an alternative is keep branch target buffer (BTB) operational and use it to obtain target addresses. In Section 5, we study the effect of different alternatives.

In addition to the more obvious benefits of reducing misspeculation and branch predictor energy, passing branch outcomes to the checker also enables other optimizations. For example, the resulting performance improvement can permit the disabling of aggressive techniques such as load speculation and dynamic memory disambiguation (through the LSQ) to further increase energy-efficiency for the same level of performance. Memory instructions can be simply issued in program order and stores can be written into the L1 cache upon execution (rather than at commit time). We examine these options in Section 5.

When a store is written to the L1 cache of a checker, its address and data are also sent to the PCB. If both the address and data match the result from the lead processor, the store entry in the PCB is marked verified (otherwise a rollback is initiated). Note that if a checker and the lead processor execute the chunk at the same time, a checker may actually run ahead of the lead processor and try to verify against an invalid PCB entry. To avoid circuit complications to handle such events, we simply delay dispatching a chunk for verification until the lead processor has finished the entire chunk. This delay between the lead and checker processors is shown in Figure 1. This intentional slack also helps to hide the communication latency in transferring the checkpoint and information in the EIQ.

Load instructions operate on the checker in largely the same way as on the lead processor. First the L1 cache and the SQ are searched. If there is a miss, in parallel with the L2 cache access, the checker searches the PCB of the lead processor (instead of its own PCB) to make appropriate updates (⑦). When a cache block is replaced from the checker's L1 cache, it is also discarded.

Finally, when the checker commits all instructions in the current chunk, it compares the architectural register file with the next checkpoint. If all the registers (including the program counter) match and there has been no mismatch in the PCB, then the entire chunk has been correctly verified. If this is the oldest chunk, the

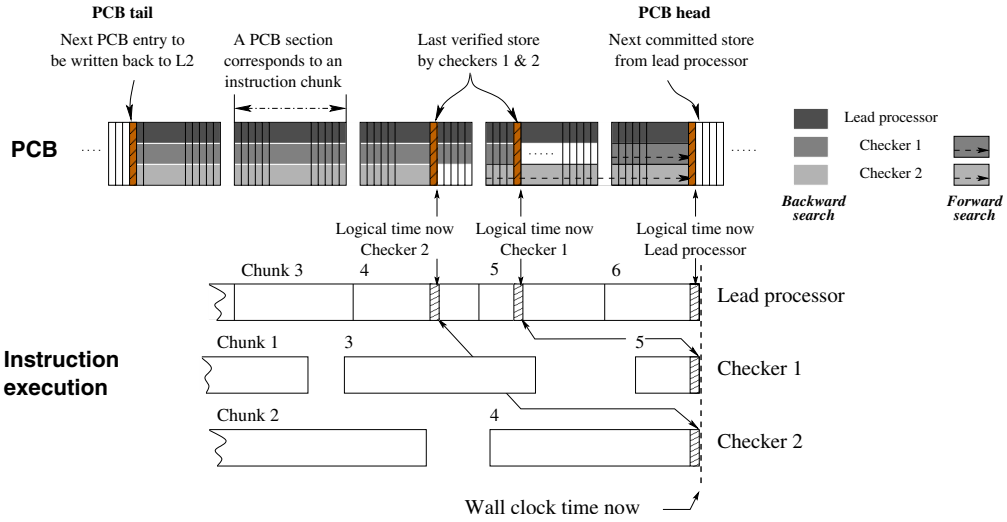

Figure 3. The different logical time frames from the different cores' perspectives. At any moment, a checker's logical time is in the "past" of the lead processor's time. Correspondingly, the checkers only see part of the memory state in the PCB (shown in different shades) in the backward search. A forward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

checkpoint representing the starting state of this chunk can be recycled. If there is a mismatch in the register contents, a signal is sent to the lead processor to roll back.

3.4 Operation of the PCB

The PCB enables rapid fault recovery, but more importantly, it is essential to the efficient creation of multiple memory images for the redundant threads. Consider the example shown in Figure 3. When 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 history" of the lead processor, and the shared memory system needs to provide the memory image as the lead processor saw it at the time it executed the same instruction. In other words, when a load executes on a checker, the combination of the SQ, the PCB, and the memory hierarchy should capture the effect of all stores prior to the load and none thereafter. With the use of the PCB, only the common part of the different images is stored in the L2 cache and the rest of the memory subsystem. The most up-to-date image is constructed on-the-fly based on which processor initiates the request. The processor's identity determines which logical time pointer of "now" to use.

Backward search To enhance efficiency, all cores use their private L1 cache to store data consistent with their view of the memory image. This makes it possible to only access the PCB when there is a miss. Thus, cache lines must contain the most up-to-date data from that processor's point of view. Therefore, when a cache line is filled, it is obtained from the L2 cache and the PCB is searched to make appropriate updates. This is called a *backward* search (see Figure 3). Note that to get an up-to-date cache line, the most recent writes to *all* words in the line need to be acquired in the backward search. While this can be done at the circuit level so that a search to PCB returns multiple words, we choose a simpler approach that only requires the circuit to 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 wrap around) and probe the PCB multiple times until we cover the entire line. This, of course, delays the availability of the cache line except the critical word.

Alternatively, to further simplify the circuit, we can limit the forwarding to the word being loaded only. Thus, if during the search, we detect the presence of other words in the same line, we only allow the caching of that word. This requires the support of per-word valid bits in the cache. Without this support, we will need to disable the caching of the entire line. Both options (especially the latter) increase the cache miss rate and thus the number of L2 accesses and PCB searches, and hence energy consumption. As we will show in Section 5, with the support of per-word valid bits, forwarding the loaded word only is a viable solution where the energy and performance degradation is acceptable. However, disabling the caching of the line completely increases the access frequency and therefore the contention in PCB so much that the performance and energy degradation becomes unacceptable.

Forward search Parallelizing the verification workload onto multiple checkers creates a peculiar cache coherence issue. When a chunk of instructions are being verified in one checker, other checkers verifying the same program will not execute the stores in that chunk and therefore may have stale data in the L1 data cache when executing a subsequent chunk. Thus, when a checker skips a chunk, we need to ensure that its L1 cache is brought up-to-date with respect to the checker's new logical time. Since by accessing the PCB we can always obtain a cache line that is up-to-date with respect to the requester's current logic time, a naive solution to this stale data problem is to force every single load

from the checkers to consult the PCB. Alternatively, we can invalidate the entire cache when a checker receives a non-consecutive chunk for verification. However, these solutions are clearly inefficient in that they introduce significant global communication between cores. Another straightforward solution involves using a state machine to walk through the sections of the PCB that correspond to the skipped chunks, to update all necessary cache lines before verifying a chunk. However, this is unlikely to be an attractive solution since such data structure traversal is usually inefficient to implement in hardware, and the resulted traffic burst may be detrimental to system performance. Finally, without modifications, the conventional cache coherence mechanism would not solve this special coherence issue as the L1 caches of different cores contain data of different times of the same thread.

To solve this problem, we implement a selective invalidation mechanism. When a chunk is skipped, since only the cache lines modified in that chunk need to be updated, we only invalidate those lines. To do so, we extend the support for the conventional invalidation-based cache coherence mechanism. The first time in a chunk the lead processor writes to a cache line (to determine which write to a cache line is the first in a chunk, we flash-clear all the dirty bits when the lead processor creates a checkpoint and starts another chunk), it sends out a quasi-invalidation message containing the cache line address and the current chunk ID to the checkers. Upon receiving the message, the checkers mark that cache line, if found in the L1, as volatile (this requires a volatile bit for every cache line), and remember the chunk ID. When a checker finishes checking one chunk and receives a non-consecutive chunk, it invalidates all the volatile cache lines associated with the IDs of the skipped chunks. Clearly, remembering the chunk IDs requires a lot of storage per cache line as each line can be modified in multiple chunks and all the IDs have to be remembered. In a degenerated implementation of this scheme, we do not differentiate between chunk IDs and thus when a checker skips a chunk, *all* volatile lines are invalidated. This scheme, however, can increase the L1 miss rate of the checkers and thus the frequency of L2 and PCB accesses. Indeed, when the workload is evenly distributed across two checkers, all the cache lines touched by the lead processor are purged out of both checkers' L1 cache.

Considering that two checkers verifying alternate chunks is a very common pattern, we choose a design that is only slightly more complex and reduces unnecessary invalidations. We assign a different color to odd-numbered chunks and even-numbered chunks (say, red and black). The quasi-invalidation messages sent include one bit encoding the current chunk color. Two bits are added to each cache line to record the color of the message. When a checker skips, say, a black chunk, only the volatile lines with the black bit set (including those with both red and black bit set) need to be invalidated.

Marking the line volatile when receiving a quasi-invalidation message alone is not enough as the line may be delivered to the cache after the message. Therefore, when a checker handles an L1 miss, it needs to obtain the volatility (and color) information. A cache line is volatile if any store to that line exists in the future chunks in the PCB. Thus, in addition to the backward search, we also search the stores belonging to the future chunks in the PCB. If there is a store sharing the same cache line address with the load, we add the volatile bit and the color information of the matching chunks in the reply. We call this the *forward search* (Figure 3).

Note that the forward and backward search are orthogonal: even if the backward search returns no match, the forward search can have a match, making the cache line volatile. Also, a volatile line is not to be confused with a non-cacheable line. A volatile line is cacheable (and cached) until the checker receives a non-consecutive chunk to verify. Finally, the support for forward search is quite simple, requiring only a cache line address match in the future chunks.

Due to soft errors, the address used in the quasi-invalidation message may be incorrect. This may cause a wrong line to be invalidated by the end of the chunk leaving the truly volatile line still in the cache. Eventually, this may cause a mismatch in the result comparison and trigger a rollback. Without removing the line from the cache, simple re-execution will only cause endless rollbacks. Hence, as mentioned before, when a rollback happens, we invalidate the entire L1 cache for all involved cores even though the data in the caches are not necessarily corrupted.

Access filtering Since the PCB is large and accessed in an associative manner, each access takes quite a bit of time and energy and thus reducing unnecessary PCB accesses is desirable. Our empirical findings show that a very significant portion of PCB searches return no hit, especially for the lead processor: on average, no more than 0.3% of lead processor-initiated searches return a match. Thus, intuitively, a quick membership test similar to that used in [19] can filter out searches that are guaranteed not to find a match in the PCB and thus reduce its access frequency.

In [19], a hash table is maintained to keep track of the number of elements in the associative queue that have a particular address hash. Any search can be avoided if the address hashes into an entry with a zero counter. However, using counters requires the handling of counter overflow [19], a complexity we want to avoid. We thus adopt a "coarse-grain" presence tracking method. We use one bit to represent the presence of an address hash in *one section* of the PCB, regardless of the exact number of instances that have the same hash in the section. This way, a single bit is enough and we will never have an overflow. Since one bit represents one section of the PCB, with K sections, each entry of our hash table is thus a presence bit vector of K bits. When a store is sent to the PCB, the cache line address is hashed to retrieve the bit vector and the bit corresponding to the current PCB head section is set to one. When any core has an L1 miss and requests a search of the PCB, the cache line address of the request is hashed to retrieve the entire bit vector, if none of the bits is set, the access to the PCB sections can be avoided. (Even if the bit vector is non-zero, the zero elements in the vector can still be used to "gate" the corresponding sections to save energy in the search.) Finally, when an instruction chunk has been verified and all the stores in the corresponding PCB section are committed to L2, we clear the bit representing that PCB section for each hash entry. To reduce "false-sharing" of hash entries, we choose a prime number to be the size of the hash table (257 in this paper).

3.5 Discussion

In terms of fault coverage, our approach can detect and recover from transient errors in the processor core, including the entire

data path and the control logic. We assume that the baseline CMP incorporates error correction codes (ECC) for the on-chip caches and main memory, as is common today. Our redundant execution strategy detects and recovers from L1 cache errors as well, which obviates the need for L1 cache ECC bits.

In our approach, checkpointing, buffering, and comparisons are largely performed outside of the core pipeline and the critical timing paths. For example, no comparison is performed in the pipeline after executing each instruction. This permits the added design complexity to be managed more easily and enables the same fault-tolerance support to be integrated into different processor implementations.

Handling of special operations There are certain types of operations that require special handling, for instance, instructions with global side effects such as I/O instructions and memory barriers. An I/O operation, even a load, can have side effects. Thus, a non-cacheable load can not be simply repeated for fault tolerance, and therefore requires the following: (a) instructions before the non-cacheable load must all be ensured to be correct to eliminate the need to re-execute the load; and (b) the result of the non-cacheable load must be buffered and fed to the trailing thread. While (b) is straightforward to implement, (a) requires forcing the lead processor to wait for the checker to catch up before executing the non-cacheable load. Support for memory barriers also requires stalling the lead processor. Clearly, for an I/O bound application or parallel workload, additional optimizations are required to make the design efficient. We leave these topics as future work and focus on single-threaded compute-intensive applications in this study.

4 Experimental Setup

To evaluate the proposed architectural support for fault tolerance, we simulate a CMP with one core running as the lead processor and various configurations of checkers. Our simulator is a modified version of the SimpleScalar [5] 3.0b tool set simulating the Alpha AXP ISA. The simulator is modified to model a CMP with the structures necessary for inter-core communication. Table 1 provides the parameters for a single core of the CMP. Our experiments are performed using the SPEC CPU2000 benchmark suite. For each application, we simulate 100 million committed instructions after proper fast-forwarding [18].

To evaluate energy consumption, we model both dynamic and leakage energy in detail. We use Wattch [3] to estimate the dynamic energy component. Our leakage energy model is temperature-dependent and is based on predictive SPICE circuit simulations for 45nm technology using BSIM3 [4]. Device parameters such as V_{th} are obtained based on 2003 ITRS projections for the year 2010, which forecasts a 45nm technology node with a clock frequency of 15GHz [7]. Temperature modeling is carried out with HotSpot [21]. We use the Compaq Alpha 21364 as a model for our proposed floor plan, microarchitecture, and power consumption. We model latency, occupancy, and energy consumption of the PCB in detail.

Fetch Queue Size	16 instructions
Fetch/Dispatch/Commit width	4 / 4 / 12
Branch Predictor	2048 entry BTB, bimodal and 2-level adaptive
Return Address Stack	32 entries
Branch Mispred. Lat.	12 cycles
Int. Units	3 ALU, 1 Mult
FP Units	3 ALU, 1 Mult
Reg. File	128 Int, 128 FP
Issue Queue	32 Int, 32 FP
LSQ / ROB	64 entries / 256 entries
L1 I/D cache	32 KB, 32B line, 2 way, 2 cycles
L2 Cache (shared)	8 MB, 32 way, 128B line, 20 cycles
TLB (I/D each)	128 entries, 8KB, fully associative
Memory latency	200 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/cycle)

Table 1. Simulation parameters.

5 Evaluation

5.1 Main Results

Except in special-purpose environments, a fault-tolerant system spends the vast majority of time in fault-free situations. For example, at sea level, cosmic particle flux is around $1/cm^2 \cdot s$ [28]. Even the most pessimistic estimation would translate this to no more than a few potential upsets per second. In terms of microprocessor clock cycles, this constitutes an exceedingly rare event. Therefore, the experimental analysis mainly focuses on the performance and energy characteristics of the design under fault-free situations.

We first evaluate the performance and energy impact of the redundant execution. We use two checkers running at half frequency to verify the computation run on the lead processor. In this part of the evaluation, whenever we reduce the frequency we also lower the supply voltage. To put the results into perspective, we show other configurations as well: using one checker at full frequency, using one checker at half frequency, the fault-tolerance mechanism proposed by Ray et al. (labeled Ray in this section) [15], which protects a smaller portion of the core, and a CRT-like design [8, 11]. In all cases, the lead processor is running at full frequency. We normalize the result to that of a single, non-redundant core running at full frequency. In Figure 4, we show the performance when turning on the fault-tolerance mechanism, while in Figure 5, we show the energy overhead.

From these figures, two observations can be made. First, in terms of performance, the degradation with two half-speed checkers is imperceptible. This is expected: although running at half-speed, the combined verification bandwidth of the two checkers is higher than the lead processor's bandwidth. Thus the lead processor rarely needs to stall because PCB sections run out. The only slowdown is caused by stalls due to checkpoint creation, occasional resource contention (*e.g.*, PCB or cache port), and cache conflicts. We also see that if the verification mechanism can not keep up (such as when running a single half-speed checker), not

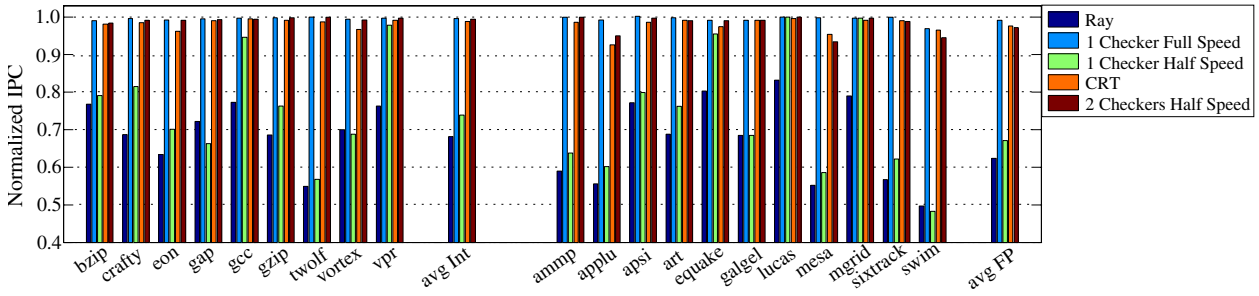

Figure 4. Performance impact of different fault-tolerant configurations.

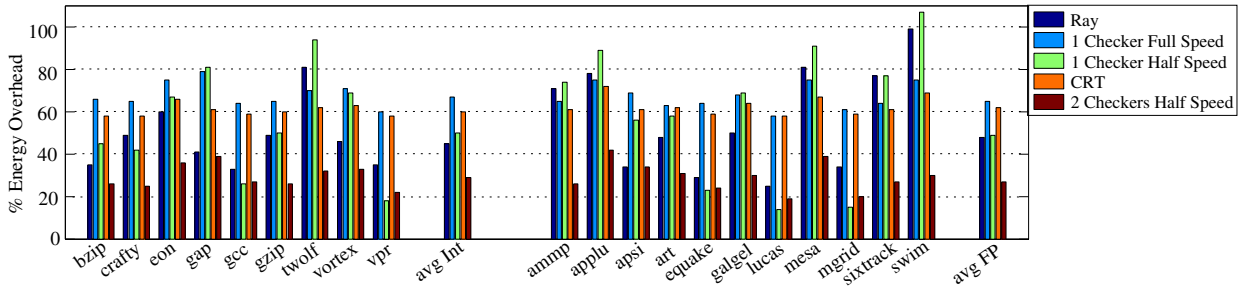

Figure 5. Energy overhead of different fault-tolerant configurations.

only performance degrades significantly, but the energy cost due to fault tolerance is also high.

Second, our design indeed improves energy efficiency of fault tolerance significantly without sacrificing performance. When using one full-speed checker, our system is very similar to a CRT microarchitecture. At about 66%, the energy overhead is similar to that of CRT as well. In contrast, when using two half-speed checkers, the overhead reduces to about 28% on average. (If we discount the energy contribution of the L2 cache, the overheads become about 110% and 40% for CRT and using two half-speed checkers, respectively.) In fact, this overhead of using two half-speed checkers is even lower than that of [15] which has a lower degree of redundancy. To further understand this result, we break down the energy consumption into different components in Figure 6. The normalized energy consumption is broken down into leakage and dynamic contributions from the L2 cache, the lead processor, and the checker(s). For clarity, we only show the average of integer and floating-point applications, as the variation from application to application is moderate.

Clearly, the two half-speed checkers introduce only a moderate amount of energy overhead. When we compare the dynamic energy component in different configurations, we see that, thanks to the reduced supply voltage, the combined energy consumption of two half-speed checkers is much less than that of a full-speed checker. The checkers' energy overhead is also lower than the increase in the core energy consumption in Ray [15] as the verification energy in that approach is also consumed on the more power-hungry full-speed processor and the extra execution time also increases the leakage and clock energy.

Overhead for rollbacks To estimate the impact of fault recovery, we compare two executions: one fault-free, the other with periodically injected errors. The execution time difference is caused by the rollbacks and the slowdown due to whole-cache invalidations. We injected errors at two different frequencies, once every 2 million instructions and once every 10 million. As expected, there is no significant difference in the per-incident cost of a rollback. The cost ranges from 1370 to 4801 cycles with an average of about 2800 cycles. Out of these cycles, about 1500 cycles on average is the delay in detecting the error. The remainder is largely the slowdown due to cache invalidation. At our assumed clock frequency, this cost of recovery translates to less than 200ns per incident. Thus, even if transient errors occur as frequently as once per milli-second, the overhead due to rollback is still minimal (less than 0.2%).

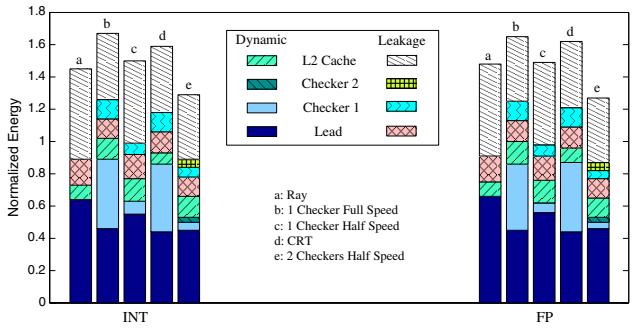

Figure 6. Breakdown of energy consumption. The configuration order of the bars is the same as in prior figures.

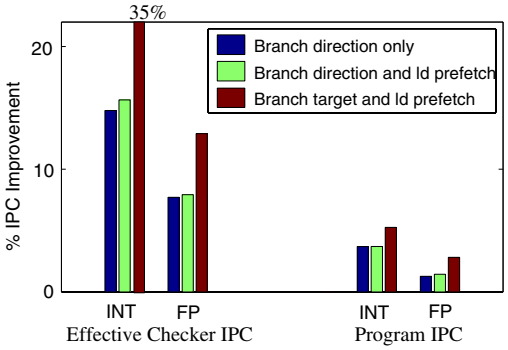

Figure 7. Performance impact of assisted execution.

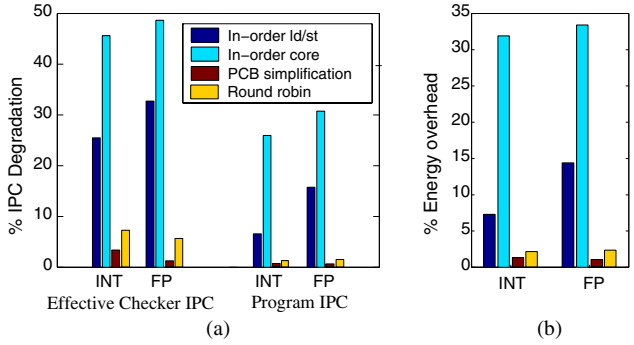

Figure 8. Performance (a) and energy (b) impact of configurations with lower complexity in executing verification workload.

5.2 Design Options and Considerations

We now discuss some of the options and considerations in the design of the architectural support. For brevity, in the quantitative analyses, we only show the average results of the integer applications and those of the floating-point applications.

Effectiveness of assisted execution The lead processor can pass on various information to assist the checkers to be more efficient. Among the choices are branch outcomes (with or without destination), and addresses for L1 prefetch. In Figure 7, we show the performance impact of these forms of assistance. In the figure, the effective IPC of the checker is that when excluding idle periods when there is no verification workload. This is a useful metric since although in our experiments the speed of the checkers is not the bottleneck, in a realistic scenario, the system may multiplex different verification workloads onto the same set of checkers, in which case the performance of the checkers may be very important. We normalize the results to a system where no information mentioned above is passed from the lead processor to the checker.

We see that with assisted execution, the checkers can be up to 35% faster on average. Information about branch outcomes is indeed very useful. Its availability improves the effective IPC by about 10%. L1 prefetch information is not as useful, improving about only 1% of the effective IPC, although it requires little bandwidth to communicate that information. Branch destination information is very helpful, especially for integer applications, improving checker performance by another 20% . 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 very small, ranging from 1% to 5%. This indicates that indeed the two checkers are not the bottleneck in this configuration and thus have processing power to spare.

Further reduction of complexity Given that the assistance from the lead processor already removes a lot of performance degrading factors, we could envision further shutting off some aggressive speculation mechanisms such as load speculation in the checkers (Section 3.3). We studied the effect of these options and other complexity reduction measures. We show their impact on the checker's effective IPC and the overall program IPC in Figure 8-(a) and that on energy in Figure 8-(b). In all cases, we normalize the results to those of our default configuration without any complexity reduction measure.

First, we see that less aggressive checkers using in-order load and store or in-order execution do not save energy. These measures have a significant performance impact. Total program slowdown ranges from 6% to 30% and is thus clearly inefficient energy-wise. We note that in our system the checkers still execute program chunks in a conventional way and experience all kinds of stalls due to dependences. This is very different from the DIVA design [1] where the main core passes the input and result of every instruction to the in-order checker and there is no dependence between instructions in the checker.

Second, as discussed in Section 3.4, to simplify the hardware, we can disable the caching of a line when the backward search detects the presence of multiple words in the cache line. However, we found that this design can result in pathological performance degradation: the disabling of caching can significantly increase the L1 miss rate for the checkers, which in turn increases PCB access frequency. In our system, the PCB has only a single search port and therefore quickly becomes a performance bottleneck. In some applications, the whole-program slowdown can be as high as 400%. With per-word valid bit support, however, we can at least cache the word being loaded. For the applications studied, the energy and performance impact of this option is negligible (less than 1%).

Third, we compare alternatives for the scheduling strategy of verification chunks. Our default strategy is a locality-conscious strategy that attempts to maximize locality by scheduling consecutive chunks to one checker and not invoking a second checker unless the PCB is more than half full. We stop a checker when there is only one chunk of instructions pending verification. Alternatively, we can simply send out chunks in a round-robin fashion. We can see from the figure, the difference between locality-conscious and round-robin scheduling is small. The difference in program-wide performance or energy consumption is only a few percent. However, the difference in effective IPC averages about 7% and can be as high as 13% in individual applications.

In summary, under our settings, none of the complexity reducing techniques achieves a net reduction in energy consumption. Using the default processor configuration with a more sophisticated locality-conscious scheduling strategy and an aggressively applied assisted execution model is the most energy-efficient ap-

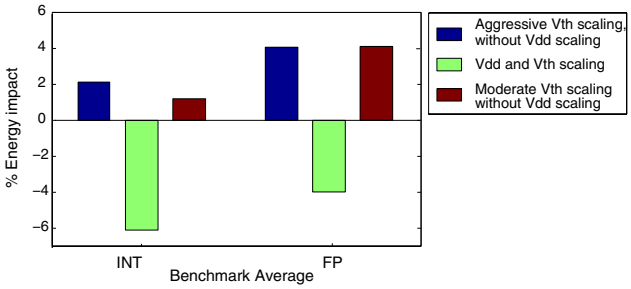

Figure 9. Energy impact of using different voltage settings for checkers at half frequency.

proach. With per-word valid bit support in the L1 cache, we can simplify the PCB forwarding logic with minimal performance and energy impact.

Adjusting voltages When we reduce the operating frequency of the checkers, we can use various voltage adjustments to increase energy efficiency: scaling down the supply voltage, which primarily reduces dynamic energy; using body biasing to increase the threshold voltage, which reduces only the leakage energy; and a combination of both for optimal energy savings. In the following experiment, we fix the operation of the checkers to half frequency and apply three different voltage configurations. First, we use body biasing to change V_{th} to 0.525V. Second, we follow Martin et al. and jointly adapt V_{th} and V_{dd} [10]. In our setup, this leads to a V_{th} of 0.31V and a V_{dd} of 0.788V. Here, we limit V_{bs} to -1V so as to avoid increasing junction leakage and the chance of junction breakdown. In our third experiment, we do not change the V_{dd} of the checkers but increase the V_{th} to 0.31V. Since the frequency is not changed, only the energy consumption of the checkers is affected. Figure 9 illustrates the overall energy impact relative to our default configuration, which only scales V_{dd} to 0.625V.

We see that indeed, joint V_{th} and V_{dd} adjustment is more effective, further reducing the energy consumption, although the magnitude of the benefit is not significant, since the energy overhead is already low (28%). We can also see that if the degree of body biasing is limited, a joint adjustment may not yield any benefit. In our experiment, it is better to just change the supply voltage. Finally, if we only use body biasing (to avoid the necessity of voltage converters between the lead processor and checkers), the additional energy overhead, at 2-4%, is tolerable.

6 Related Work

Thread-level redundancy is not a new concept. Many commercial fault-tolerant computers such as the IBM z900 [22] have used fully duplicated hardware operating in lockstep. Recently more decoupled forms of TLR have been proposed [8, 11, 16, 17, 27]. In this prior work, the energy cost of TLR is not addressed. Given the rising need for energy-efficient computation, our approach is to exploit the inherent parallel nature of verification by performing this task across multiple checkers with more energy-efficient configurations. This requires a much higher degree of decoupling between the leading and trailing threads than prior work. While these prior TLR designs are limited by the number of uncommitted stores, our speculative L1 cache and PCB provide an efficient memory buffering mechanism that can hold a large amount of unverified stores and enable fast search and data forwarding. This allows the slack between the lead and checkers to be far greater than in these prior proposals, which creates a large enough verification workload to operate two parallel checkers at half speed without undue performance loss.

Other efforts provide error detection and recovery through dedicated architectural support embedded in the processor pipeline. Austin's DIVA approach uses a second in-order checker unit which re-executes instructions coming from the out-of-order execution engine [1]. DIVA exploits the verification parallelism at a much finer granularity: at the instruction level. Ray et al. use the superscalar execution mechanisms to execute two copies of the same instruction to detect and correct errors primarily in the data path [15]. Parashar et al. employ an instruction reuse buffer in order to reduce the performance cost of Ray's approach [12]. Smolens et al. evaluate resource sharing in such instruction-level redundancy approaches [24]. Our approach, by contrast provides core-level redundancy in a way that is most similar to chip-level redundant threading (CRT) [11] and similar approaches [8], in which a multi-core processor can be harnessed for fault tolerant operation with additional hardware support, largely outside of the core pipeline. Our work is also largely orthogonal to efforts of creating globally-consistent checkpoint state for roll-back in multiprocessors running parallel programs [13, 25].

Purser et al. evaluated alternative memory hierarchy designs of Slipstream [14]. In one design, they also allow the A-stream's speculative data to be committed to L1 cache and discarded when replaced. Without a PCB-like structure in their design, this results in incorrect data in the L1 cache when a displaced dirty line is re-accessed and would cause the A-stream to manifest a misspeculation. Their insight is that the trailing R-stream is not far behind and therefore the amount of incorrect data is small and can be tolerated since the cost of a recovery in Slipstream is still reasonable. With a much more deeply-decoupled execution and at a cost of close to 3000 cycles per roll-back, our system can not efficiently tolerate such speculation. Our buffering mechanism is different in that our L1 state is speculative only with respect to soft errors. Without soft errors, it is completely non-speculative.

Finally, master/slave speculative parallelization also uses the model of a main thread (master) followed by "checkers" (slaves) each executing a subset of the program [29]. The purpose is to speed up speculative parallelization and the master executes a compiler-generated speculative version of the program.

7 Conclusions

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 execution is a straightforward and effective mechanism for both fault detection and recovery. However, dedicated redundancy is not only inflexible but also 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 met.

We propose a design that matches the multiprocessing capa-

bilities of modern microprocessors with the inherent parallelism in correctness verification. Our approach requires modest architectural support, uses identical processing elements, and leverages these capabilities to provide a flexible platform for redundant execution. Fault tolerance can be provided on demand based on the criticality of applications and delivered in an energy-efficient manner, by exploiting the natural parallelism of verification. More importantly, the new mechanisms that we introduce to deeply decouple the lead and checker operations permits several checkers running at a slower, lower energy operating point to be employed without undue performance loss. Through a detailed evaluation, we demonstrate how the use of two half-frequency cores for redundant execution achieves efficiency that rivals that of approaches with more limited protection.

References

[1] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. In *International Symposium on Computer Architecture*, pages 26–37, June–July 2001.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] BSIM Design Group, World Wide Web, `http://www-device.eecs.berkeley.edu/~bsim3/ftv322/Mod_doc/V322manu.tar.Z`. *BSIM3v3.2.2 MOSFET Model - User's Manual*, Apr 1999.

[5] D. Burger and T. Austin. The SimpleScalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison Computer Sciences Department, Jun 1997.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *International Symposium on Computer Architecture*, pages 186–195, May 1999.

[7] Executive Summary. *The International Technology Roadmap for Semiconductors*. World Wide Web, `http://public.itrs.net/Files/2003ITRS/ExecSum2003.pdf`, 2003.

[8] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *International Symposium on Computer Architecture*, pages 98–109, June 2003.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[10] S. Martin, K. Flautner, D. Blaauw, and T. Mudge. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors Under Dynamic Workloads. In *International Conference on Computer-Aided Design*, pages 721–725, Nov. 2002.

[11] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *International Symposium on Computer Architecture*, pages 99–110, May 2002.

[12] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *International Symposium on Computer Architecture*, pages 376–386, June 2004.

[13] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, pages 111–122, May 2002.

[14] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream Memory Hierarchies. Technical report cesr-tr-02-3, Department of Electrical and Computer Engineering, North Carolina State University, 2002.

[15] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *International Symposium on Microarchitecture*, pages 214–224, Dec. 2001.

[16] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture*, pages 25–36, June 2000.

[17] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.

[18] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical Report RC-21852, IBM T. J. Watson Research Center, Oct 2000.

[19] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, Dec. 2003.

[20] P. Shivakuma, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[21] K. Skadron, M. Stan, W. Huang, S. Velusamy, and K. Sankaranarayanan. Temperature-Aware Microarchitecture. In *International Symposium on Computer Architecture*, pages 2–13, June 2003.

[22] T. Slegel, R. A. III, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, Mar./Apr. 1999.

[23] J. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.

[24] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitecture. In *International Symposium on Microarchitecture*, pages 257–268, Dec. 2004.

[25] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, pages 123–134, May 2002.

[26] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[27] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *International Symposium on Computer Architecture*, pages 87–98, May 2002.

[28] J. Ziegler et al. IBM Experiments in Soft Fails in Computer Electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, Jan. 1996.

[29] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.