Supporting Highly-Decoupled Thread-Level Redundancy for Parallel Programs[*]

M. Wasiur Rashid and Michael C. Huang
Dept. of Electrical & Computer Engineering
University of Rochester
{rashid,huang}@ece.rochester.edu

Abstract

The continued scaling of device dimensions and the operating voltage reduces the critical charge and thus natural noise tolerance level of transistors. As a result, circuits can produce transient upsets that corrupt program execution and data. Redundant execution can detect and correct circuit errors on the fly. The increasing prevalence of multi-core architectures makes coarse-grain thread-level redundancy (TLR) very attractive. While TLR has been extensively studied in the context of single-threaded applications, much less attention is paid to the design issues and tradeoffs of supporting parallel codes. In this paper, we propose a microarchitecture to efficiently support TLR for parallel codes. One of the main design goals is to support a large number of unverified instructions, so that long latencies in verification can be easily tolerated. Another important objective is to have a comprehensive coverage that includes not only the computation logic but also the coherence and consistency logic in the memory subsystem. Hence, the redundant copy of the program needs to independently access the memory and the system needs to *efficiently* manage the non-determinism in parallel execution. The proposed architectural support to achieve these goals is entirely off the processor critical path and can be easily disabled when redundancy is not requested. The design, with a few effective optimizations, is also efficient in that during error-free execution, it causes less than 3% additional performance degradation on top of throughput loss due to redundancy.

1 Introduction

As the feature size of VLSI technology continues to shrink and the operating voltage continues to decrease, the noise tolerance of individual circuit elements fundamentally decreases. Consequently, the circuit is more vulnerable to unwanted energy such as that from the impact of a particle. In some cases, the reduction in a transistor's critical charge can dramatically increase the error rate of certain noise mechanisms [18]. At the same time, higher levels of integration increases the number of on-chip noise sources as well as the total number of devices (potential victims) on a chip. All in all, the frequency of transient (or even permanent) errors in future generations of microprocessors, if unprotected, will increase.

Although for certain users, hardware errors represent a small, perhaps unnoticed portion of causes for system failures, dwarfed by more mundane errors such as software bugs and mis-configurations, they are nonetheless real and significant concerns, especially for server systems where the software is treated with much more rigor and diligence, and the systems tend to work continuously with much less idling. Highly publicized incidents of soft-error-related failures in commercial products [10, 24] underscore the importance of fault tolerance. Therefore, it is crucial to develop effective mechanisms, especially on-demand ones that offer users the choice of protection (at the cost of throughput and energy). Most modern processors already heavily rely on error-correction codes to protect storage elements such as caches and pipeline registers [2, 40]. However, they typically leave the logic unprotected. In the general-purpose domain, while a user can now select machines with a highly reliable memory system [13], there is little choice to boost the reliability of the logic. Studies have shown that protecting against errors in the core logic is important and will be increasingly so in the future [41, 42]. With the exception of ALUs where using a checksum is possible [33,55], brute-force redundancy is perhaps indispensable for random logic, even just to *detect* transient errors. As such, the key practical issue really becomes how and at what level to apply redundancy.

A very significant body of work explored redundancy at the circuit level (*e.g.*, [19, 30]). While these solutions can mitigate the problem, there are *fundamental* limitations (see Section 2). Furthermore, redundancy is undoubtedly expensive and ideally is only provided on-demand when the protection is necessary. At best, it is cumbersome to disable fine-grain circuit-level redundancy once it is in-place. In contrast, *Thread-Level Redundancy* (TLR) avoids these disadvantages by using the increasingly abundant extra hardware thread contexts to execute a semantic thread redundantly. While TLR for single-thread applications has been extensively studied [3, 17, 27, 31, 36, 37, 39, 45, 53], design tradeoffs of TLR for parallel applications received much less attention, while parallel applications are bound to become more numerous. TLR support for parallel applications is not a trivial extension of that for single-threaded applications and has a large design space to be explored.

In this paper, we present the architectural support for a chip multiprocessor (CMP) to enable on-demand redundant execution of parallel applications in a highly-decoupled fashion. The design is decoupled in two aspects. First, rather than (quasi) lock-stepping, the two redundant copies of the same semantic thread are highly decoupled in time and can be thousands of instructions apart. Second, the architectural support for managing redundancy (such as buffering results from unverified instructions) is decoupled from the core microarchitecture and the core is virtually redundancy-oblivious. This makes redundancy easy to turn on or off. When not engaged, the support exerts no overhead in energy or performance. Even when it is enabled, the additional performance overhead incurred is only 2.5% on top of throughput loss due to TLR.

The rest of this paper is organized as follows. Section 2 discusses high-level tradeoffs in redundancy and provides an overview of the design. Section 3 details the design. Section 4 summarizes

[*] This work is supported in part by the NSF under grants No. 0509270 and 0719790.

the experimental setup. Section 5 provides the experimental analysis. Section 6 discusses related work. Finally, Section 7 concludes.

2 High-Level Design Decisions

Redundancy as a means to tolerate random errors is a classic technique [29]. In an actual design, the decisions are driven by a multitude of sometimes conflicting desires. We first discuss the high-level tradeoffs made in our proposed design.

Granularity of hardware redundancy A very significant body of work has explored redundancy at the circuit level (*e.g.*, [8, 15, 19, 26, 32, 55], to name but a few). While these solutions can mitigate the problem, there are fundamental limitations.

First of all, a pivotal assumption of redundant designs is that errors occur independently. For this to work in a fine-grain spatial redundancy such as duplicating critical transistors [19], the spatial distance between redundant nodes has to be "sufficiently" large. Otherwise the nodes can be upset simultaneously in a correlated fashion and errors will thus be *systematically* undetected. The actual "safe" distance depends on the noise mechanism and can be hard to quantify. But studies suggest that it is non-trivial. For instance, the spatial range of electron-hole pair generation after a particle impact is about $0.1\mu m$ [62]. When diffusion is taken into account, a pair of redundant transistors have to be physically far apart (several microns [25]) to ensure they are not impacted the same way. Such physical layout constraints make the technique very hard to scale in future generations. Furthermore, such spatial redundancy is not without time overhead: significant amount of time (substantially larger than 50ps [19]) is needed for the redundant node to "fight" the upset node to set the output right.

Secondly, time redundancy at circuit level is also a limited solution. The fundamental property exploited here is that a static combinational circuit will naturally recover from transients since they last only a limited amount of time. The output of the logic can therefore be strobed by the latch at different time points separated by a fixed amount t. Any transient pulse with a width narrower than t will at most affect one such latched result. A majority vote circuit can find the correct value out of 3 different latched results [25, 30]. With such techniques, the cycle time needs to increase by at least twice the maximum width of the transient pulse to be tolerated. Unfortunately, such transient pulses are generally wide. Experiments have shown many of them caused by particle strikes to be more than 800ps wide [8] – several times the size of clock cycle of modern processors. Clearly, the time overhead is unacceptable in high-end microprocessors. Just as space-redundant designs also have time overhead, these time-redundancy approaches add space (and power) overhead in addition to the time overhead. Furthermore, such time redundancy can not be directly applied to dynamic circuits.

To overcome these fundamental limitations, we choose a coarse-grain redundancy – TLR. In TLR, redundant copies of the same semantic thread execute on different hardware contexts and compare the architectural results. In this paper, we use CMP as the underlying base architecture. We execute two redundant copies of every semantic thread. These threads are grouped into a *computing wavefront* followed by a *verification wavefront* as illustrated in Figure 1. We compare the outcome from the two wavefronts to detect transient errors and roll back to an earlier checkpoint for recovery.

This coarse-grain redundant system avoids these aforementioned limitations and offers a much more comprehensive error detection coverage. It also provides a range of additional benefits as flexibility and intelligence can be more readily built into a coarse-grain redundant design.

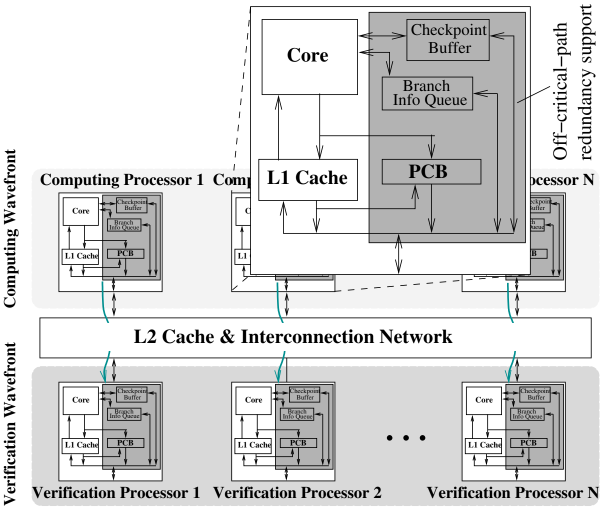


Figure 1. Illustration of the proposed TLR design.

First, coarse-grain redundancy is much less intrusive than localized fine-grain redundancy which often introduces logic to critical paths; adds design complexities that can impact schedule; and is not suited to full-custom design [44]. Second, it is much easier to provide on-demand redundancy. A single core can provide redundant execution or simply act as a normal computing engine. Finally, upsets at fine granularity do not necessarily result in architecturally visible errors due to all kinds of masking or dead state [54, 56]. These errors are naturally ignored in TLR, whereas in a fine-grain redundancy approach, it is not straightforward for the circuit to make the distinction.

Coverage A key design tradeoff in TLR is whether to cover the logic in the memory subsystem, or, using the terminology in [37], whether to include it in the *sphere of replication*. It is our belief that this logic needs to be protected in an integrated way together with the processor cores, especially when supporting parallel applications.

In current microprocessors, to correctly service load and store instructions, the memory subsystem incorporates non-trivial amount of logic in the complex memory dependence, coherence, and consistency handling logic in the cache and in the load-store queues inside the processor core [11, 43, 52]. Error-correction codes can only protect storage, not the logic in these circuits. Given that memory instructions are very frequent, leaving this logic unprotected is perhaps unwise.

Simply replicating the core does not automatically protect every logic block. We need to ensure cores independently (and redundantly) exercise the replicated logic. For example, if we rely on one core to load data and duplicate it for the redundant core, we essentially bypass the redundancy for memory logic blocks and leave them unprotected. As a result, even though the circuitry is apparently replicated, a range of errors will be undetected. For example, if in response to an invalidation request, the coherence logic invalidated the wrong cache line; if the load-store queue incorrectly forwarded data or failed to detect a load-store order violation; or if a memory barrier is incorrectly enforced, the core may consume incorrect data even though the integrity of the cache storage is not compromised.

Based on these considerations, in our architecture, the two wavefronts independently access their own *logical* memory subsystem. Cache coherence is maintained within each subsystem. In reality, duplicating the entire memory subsystem is wasteful and the two logical subsystems share some portions in the physical implementation (L2 and beyond, in this paper).

Lock-stepped or asynchronous redundancy Another design decision is whether to make the redundant threads progress in a cycle-by-cycle lock-stepped fashion or allow them to proceed asynchronously (even maintaining an intentional distance). While lock-stepping offers conceptual simplicity, it is increasingly difficult to ensure [4]. Any non-determinism (such as in cache replacement) or harmless discrepancy (such as bit-flips in prediction tables) can easily make a processor pair lose synchrony. Resynchronizing them requires initializing even the microarchitectural state to be the same. This process is not only slow, but demanding to implement.

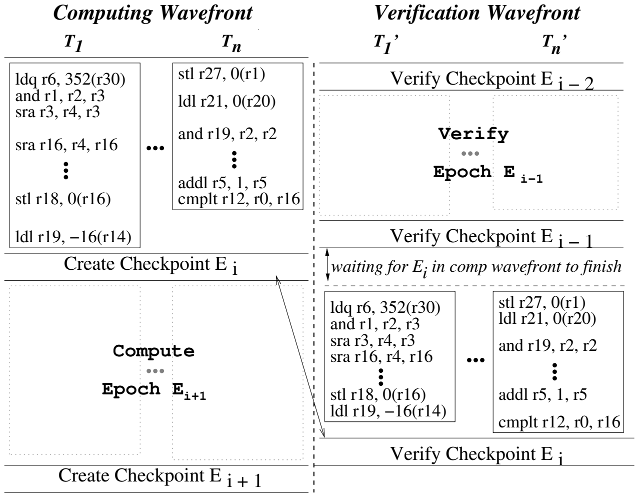
Allowing the threads to be decoupled, on the other hand, tolerates unimportant divergence of microarchitectural states. It can also tolerate latency to communicate and compare results and state between the redundant threads. It is also possible to leverage the leading thread's essentially near-perfect program-based prefetch and branch predictions for the trailing thread [37]. This, in turn, may allow the trailing thread to execute on a low-power state (*e.g.*, lower frequency/voltage gear) or on a less power-hungry microarchitecture (if heterogeneity [20] is supported) without slowing down the entire program.

In our design, therefore, we choose the decoupled, asynchronous redundancy where the leading wavefront is considered the *computing* wavefront and the trailing one, the *verification* wavefront. As illustrated in Figure 2, we compare the state of the two wavefronts periodically at *epoch* boundaries (typically thousands of instructions per thread). Certain conditions will trigger the termination of an epoch for the entire computing wavefront. Each processor then creates a checkpoint (or snapshot) of the architectural state to allow rollbacks. Even though epoch termination is a globally synchronous event, the performance overhead is not as high as one might expect since only the commit stage temporarily freezes. For the verification wavefront, the operation is largely the same except two things. First, epoch boundaries are determined by the computing wavefront. Second, at the boundaries, the processor will need to compare checkpoints from the two wavefronts.

As a result of this decoupling, at any time, the number of unverified instructions can be large, perhaps thousands or more per thread. Therefore, we need a more scalable buffering mechanism than the rather limited buffering from the out-of-order engine. Our solution is to use a separate buffer to keep committed, but unverified, stores.

Putting it together In our TLR design, we execute the program in two wavefronts (computing and verification) independently as if they are executing two different programs (albeit with the same executable and input, and they only produce one output).

- Both wavefronts independently access their own memory subsystem. (However, some portion of the memory is shared as described in Section 3.1.) Coherence activities in one wavefront do not affect the other wavefront.
- Periodically, the two wavefronts compare architectural state to detect discrepancy caused by errors. Of course, inherent non-determinism in parallel execution has to be suppressed for this to work. The verification wavefront will maintain the same memory access order as the computing wavefront (Section 3.2). As a result, the trailing verification wavefront will even reproduce the same number of iterations in an idle loop.



```
ldq r6, 352(r30)
and r1, r2, r3
sra r3, r4, r3

sra r16, r4, r16
⋮
stl r18, 0(r16)

ldl r19, −16(r14)
```

```
stl r27, 0(r1)

ldl r21, 0(r20)

and r19, r2, r2
⋮
addl r5, 1, r5
cmplt r12, r0, r16
```

```
ldq r6, 352(r30)
and r1, r2, r3
sra r3, r4, r3
sra r16, r4, r16
⋮
stl r18, 0(r16)
ldl r19, −16(r14)
```

```
stl r27, 0(r1)
ldl r21, 0(r20)

and r19, r2, r2
⋮
addl r5, 1, r5

cmplt r12, r0, r16
```

Figure 2. The operation of the TLR architecture: the computing and verification wavefronts and the epochs. Due to branch predictions and prefetching provided by the computing wavefront, threads in the verification wavefront have different timing and are generally faster.

In this paper, our discussion will focus on one design point: a CMP baseline architecture with private L1 caches (with an invalidation-based snoopy protocol); a shared L2 cache; and a sequential consistency model. This represents perhaps the most straightforward case to support TLR for parallel applications. Extending the support beyond this baseline is our future work. Finally, we do not address the issues of supporting I/O in TLR.

3 Architectural Support

We first discuss the mechanisms to buffer and compare architectural states (Section 3.1) and then discuss the coordination of memory access ordering in the two wavefronts (Section 3.2), as the latter depends on the former. Note that all the supports are off the critical path; each processor can be easily configured to participate in either wavefront or to operate non-redundantly as a regular processor.

3.1 State Buffering and Comparison

Versioning versus buffering Because the two wavefronts are asynchronous, at any moment, their semantic view of the memory is different from one another. The trailing verification wavefront sees an image of the memory that the leading computing wavefront saw some time in the past. Therefore, the TLR system needs to provide data based on which wavefront the requester belongs to. In a sense, a cache line has (at least) two versions. Furthermore, upon a recovery, the memory system needs to restore to an earlier state, essentially requiring additional versions to be kept.

We do not use the caches to explicitly deal with multiple versions as such version management (forking and merging versions) will certainly be intrusive and impact critical paths. On the other hand, keeping two sets of completely separate memories is wasteful. In our design, we avoid the need of explicit versioning by keeping only validated data in the shared portion of the memory hierarchy (the L2 cache and beyond, in our case) since there is only one version of validated data for both wavefronts[1]. As such, unverified data need to be buffered in the memory hierarchy private to the cores.

3.1.1 Post-commit buffer

To avoid intrusive designs, we decouple the redundancy-incurred buffering from normal instruction processing. The core commits instructions as usual without waiting for verification. A *post-commit buffer* (PCB) keeps committed stores until verification. Although the concept of PCB was introduced in our earlier work [35], to support shared-memory parallel execution, the design discussed here is very different in a number of aspects and works as follows.

When a store is committed, the data is written into *both* the L1 cache *and* the PCB. The PCB is partitioned into sections, corresponding to epochs. In the computing wavefront, when one processor's current PCB section fills up, a new epoch is initiated and all processors start to use the next section of their PCBs. When an epoch is validated, the content will be committed to the L2 cache and that PCB section will be reused for a future epoch. The PCB is cache line-based and each section is essentially a tiny fully-associative cache (other associativity is also possible).

The PCB keeps all the dirty lines and eventually writes them back after validation. The L1 cache no longer does writeback. A dirty cache line evicted by the L1 is simply discarded. Thus, the PCB is effectively an extension of the L1 cache and needs to be treated accordingly. For example, when an access misses the L1 cache and the PCB, we need to search other processors' L1 cache and PCB in addition to accessing the L2 cache. Likewise, when one processor sends an invalidation, it needs to be applied to the lines in the PCB as well.

One important difference between the PCB and a normal cache is that the PCB is divided into sections corresponding to epochs and thus if a cache line is written to in two epochs, the same line will appear in two PCB sections. This introduces a state management issue illustrated in Figure 3.

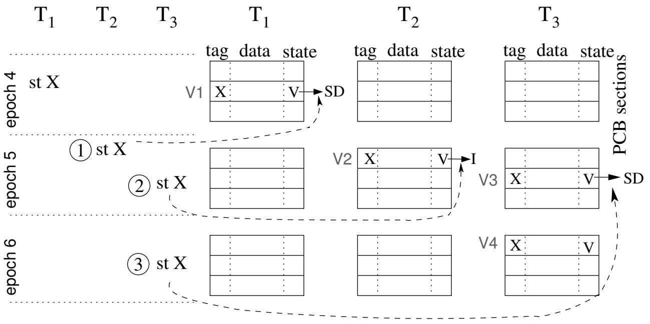


Figure 3. Example of different versions and states of PCB lines.

In this example, cache line X is written to in epoch 4 by thread T1, and later on in epoch 5, first by T2, then by T3, and again by T3 in epoch 6, creating 4 different versions in the PCBs with different roles. V4 is clearly the most up-to-date version and will be supplied to a requester if cache line X is evicted from the L1 cache of T3. The older versions V1 to V3 no longer need to respond to such requests. However, there is a difference between them. V1 is the most up-to-date version in epoch 4. When epoch 4 is finally validated, V1 needs to be written back to L2 so that the system has the ability to roll back to the beginning of epoch 5. V2, on the other hand, is no longer needed and its PCB entry can be recycled.

Thus, the PCB cache line can be in three different states: *Valid* (*e.g.*, V4), where the data can be supplied to a requester and will be written back to the L2 upon validation; *Superseded* (*e.g.*, V1) where the data will not be supplied to any read request, but will be written back; and *Invalid* (*e.g.*, V2), where the storage in the PCB is no longer needed and can be used for another cache line.

The PCB responds to coherence messages as follows. When an invalidation message is received, any valid matching line in the earlier PCB sections will be marked as superseded (*e.g.*, V1 as T2 executes the store ①), whereas the matching cache line in the current section will be marked as invalid (*e.g.*, V2 as T3 executes the store ②). A line can also be superseded when the same processor writes to it again in a new epoch (*e.g.*, V3 as T3 writes again in epoch 6 ③). When a read request is sent to a processor, if the L1 does not contain the data, all the PCB sections are searched. If there exists a matching line with valid state, the line is supplied to the requester.

Note that an important advantage of using these three states, especially in a snoopy coherence protocol, is that out of all the PCBs and all the sections within one PCB, at most one copy of any particular cache line can be in valid state. This means there is no need to arbitrate the right version among multiple responders and also that the PCB does not need a priority encoder logic to arbitrate the right version among the multiple sections – these supports are needed if we do not have the superseded state. Furthermore, after validation, when the content of the PCB is being committed to the L2 cache, multiple PCBs can do so in parallel since within any epoch, there is only one version of a cache line in either valid or superseded state.

Finally, during rollback, we can easily discard the writes done in an epoch by treating its PCB section as invalid. For the L1 cache, however, it is not easy to identify lines that need to be invalidated. Walking over the current PCB section and invalidate the corresponding lines in the local cache is not enough. We need to also send invalidations to remote caches in case they have a copy. We avoid this complexity and the concentrated invalidations and simply invalidate all cache lines.

3.1.2 PCB optimizations

When a PCB section is filled up and can not accommodate another store, a new epoch needs to be initiated. Overly frequent epoch initiation obviously adds overhead to the execution. Thus, PCB sections are fully associative to efficiently utilize its capacity. To reduce the frequency of associative tag-checking, we apply two optimizations: the *index pointer* and the *bloom filter*.

Index pointer The index pointer points from a line in the L1 data cache to its "shadow copy" in the PCB. After the pointer is established (*e.g.*, cache line X in Figure 4), subsequent writes to the same cache line only need to follow the pointer. Without the pointer, every write needs an associative search of the PCB to find the line to write to[2]. The pointer is established upon the first write to a

[1] Note that this leaves the L2 and beyond to other means of protection as they are not covered by our TLR.

[2] This observation may seem to suggest that updating the PCB for every store is undesirable to start with. However, the alternative of only moving entire cache lines from the L1 to the PCB is more complex. For instance, it requires a walk logic to move multiple cache lines at epoch boundaries. In

cache line in each epoch and may be re-established after eviction and refetch. If a write occurs to a dirty (modified) line pointing to an older PCB section (*e.g.*, the dashed pointer of cache line Y in Figure 4), then the current write is the first write in this epoch and a new entry in the PCB will be allocated to store a copy of the cache line. Before we update the pointer, we follow the old pointer and set the entry's state to "superseded".

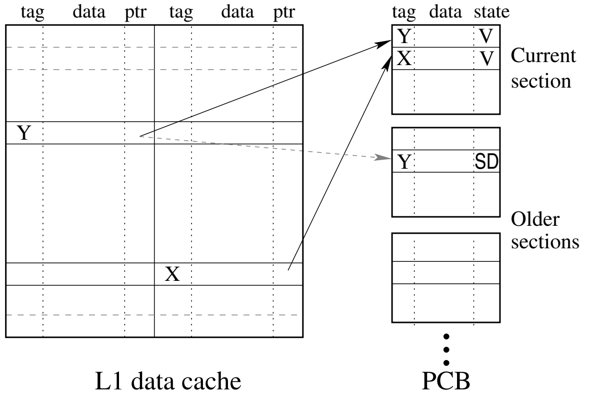


Figure 4. Illustration of the index pointer from L1 cache to the PCB. **V** and **SD** stand for valid and superseded, respectively.

If a cache line is evicted and later reaccessed, the pointer will be re-established: as we serve the L1 miss, the PCB is searched, and if a match (to a valid line, not a superseded line) is found, the pointer will be set to link the L1 version to the PCB version. Thus, if a cache line has a null pointer, we know that no such line exists in the PCB in valid state. Then when a write occurs to that line, we proceed directly to allocate a new entry in the PCB. Note that updating the pointer of a cache line is done either when servicing a cache miss or when servicing a write operation. Nothing is done on the critical path of a cache read.

This pointer-setting algorithm guarantees that for every line in the cache, either there is no valid copy of the line in the PCB (pointer is null), or we have the exact location of the PCB copy. Therefore, when an external invalidation message arrives and matches a line in the L1 cache, we do not need to associatively search the PCB. Instead, we only follow the pointer (unless it is null) to set the PCB copy to invalid (if it is in the current section) or superseded (if it is in an older section).

Similarly, the pointer also helps avoid searching the PCB associatively when processing a read request from another processor. Specifically, the saving occurs when an external read request finds a matched line in the L1 cache and the line is in the *shared* state. Clearly, if the line is in the modified state, the line will be supplied to the requester and there is no need to further probe the PCB, even without the pointer. However, if the line is in the shared state, the situation is different from that in a conventional protocol where no forwarding is needed. Consider the following case: a processor (p_i) writes to a line, and then a subsequent read request from another processor downgrades the line to the shared state. Unlike in the conventional system where during this downgrade, the data is written back to the next level of the memory hierarchy, we can not write the data until after the validation of the epoch. Therefore, in this case, p_i is still the "owner" of the data and its PCB may be the only place where this data is kept. As such, the PCB needs to be consulted in order to know whether p_i is the owner or just a sharer. In the former case, the line will be present in the PCB and in valid state. With the indexing pointer optimization, we do not need to search the PCB associatively. Instead, following the pointer is sufficient.

As PCB sections are being recycled, the indexing pointers can become stale. Stale pointers can be proactively invalidated as a PCB section is recycled. As each PCB entry has the entire address, we can follow back to the L1 cache and change the pointer to null if needed. Alternatively, the pointers are not proactively updated, but a sanity check is performed whenever the pointer is dereferenced and set to null if it points to an unrelated line.

Finally, we note that although conceptually the pointer is part of the L1 cache, physically, the mechanism can be implemented outside the cache to minimize intrusion, as following the pointer to access the PCB is always independent to the L1 access.

Bloom filter Even with the support of the pointers, the PCB may still need to be searched. In a broadcast-based coherence protocol, coherence messages will be quite frequent and so will PCB searches. The often-used bloom filter can cut down on the number of actual PCB searches. Because we recycle the PCB entries section by section, we use one filter for each section so we can simply clean up the corresponding filter when the section is recycled.

Thanks to the indexing pointer, we can now set the bloom filter in a unique way that reduces the probability of false positives. As we will see later, this optimization significantly reduces the number of bits set in the bloom filter and drastically improves filtering efficiency.

For convenience of discussion, we will call a (valid) line in the PCB that is also currently in the L1 data cache a *mapped* line. Recall that if a coherence message finds a match in the L1 cache, we only need to follow the pointer. Also, a locally incurred PCB search will be done only during a cache miss. In other words, we will never search the PCB with the address of a mapped line. Conversely, the mapped lines will never match any associative search. Thus, for the purpose of associative searches, we can consider the mapped lines as not even present in the PCB. As a result, we do not need to set their corresponding bit in the bloom filter – we only do so when a mapped line is evicted from L1 and the pointer still points to the right copy. Additionally, the PCB lines have an explicit *mapped* bit to track which lines are present in the L1. With these bits, the mapped lines (as well as superseded lines) can avoid actual address comparison even when the bloom filter fails to prevent an associative search.

Conserving writeback bandwidth Partitioning the execution into epochs creates an undesirable side effect. A cache line is temporally separated into multiple versions, each individually committed to the L2 cache when the epoch is validated. This can increase the L2 traffic and contention. We use a simple mechanism to mitigate this increase. The key observation is that if a PCB line in an earlier epoch i is superseded by another line in a later epoch j, then we can avoid the writeback of i's version if epoch j is validated. Therefore, we can simply delay committing PCB's result to the L2. The longer we wait, the more likely we find superseded lines. However, the core in the computing wavefront can not reuse a PCB section until the same section has been committed to the L2 cache. Thus, excessive delay in writeback can cause stalls in the computing wavefront and result in performance degradation. Our approach is to send a signal from the computing processors when

contrast, simultaneously updating the PCB is a background action incurring little interference to the timing-critical L1 cache.

available PCB entries are below a certain threshold. Upon receiving the signal, the verification processors start to release the oldest PCB section.

3.1.3 Comparison of state

Our TLR support compares the outcome of threads at the granularity of epochs (thousands of instructions per thread). At this granularity, the register state is best compared in its entirety, whereas for memory, we should compare the incremental changes made during the epoch, *i.e.*, the content of the PCBs. Our design can buffer thousands of unverified instructions (see Section 5) and therefore can easily tolerate the latency to transmit and compare large amounts of data. The results an epoch produces include register content, content of the valid lines in PCB, and a small amount of meta data (such as the number of valid lines in the PCB). This is on the orders of 1KB per pair of threads per epoch in a typical configuration (Section 5).

To cut down the communication bandwidth requirement, one can use a checksum (*e.g.*, CRC) to compress the state [46]. Once the checksums for both wavefronts match, the epoch is successfully validated. We can start to commit the writes buffered in the PCB to the L2 cache. As mentioned earlier, within each epoch, no cache line will be present in more than one PCB, so the write-back process can be done in parallel. Because of the redundancy, only one wavefront needs to write back. We choose the trailing verification wavefront to minimize any bandwidth impact of the write-back on the leading computing wavefront. Intuitively, by trailing behind, the verification wavefront receives natural prefetch benefit and can accept branch outcome from the leading computation wavefront [37] and is thus not the performance bottleneck.

Note that the comparison of outcomes (with compression or otherwise) is entirely a background process. The latency only dictates when PCB sections can be written back and thus released.

3.2 Memory Access Ordering Issues

Allowing asynchronous progress of the computing and verification thread in sequential workloads is relatively straightforward. However, in the case of parallel shared-memory programs, the two wavefronts typically have very different execution timing. This is because both cache misses and mispredictions are significantly mitigated in the verification wavefront. The difference in execution timing means that memory access races (including synchronization and data races) can play out differently and result in different execution outcomes without any transient errors. The system will unnecessarily roll back. Worse still, rollback itself does not address the problem of non-determinism, and can not guarantee forward progress.

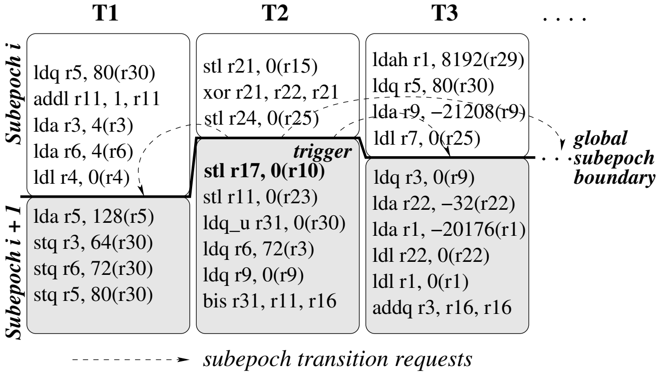
Thus, a key aspect of any TLR architectural support for parallel codes is to manage this non-determinism. There are two possible approaches: to tolerate non-determinism in the races and handle the fallout when it does lead to discrepancy; or to eliminate the non-determinism in the first place and ensure all the races always play out exactly the same way in both wavefronts. There are pros and cons for each strategy. The latter is more direct and perhaps conceptually easier to implement, but there are potential disadvantages on the system's efficiency. First of all, strictly enforcing certain access ordering can induce excessive waiting, especially in our TLR system, where the "natural" execution timing from the two wavefronts can be very different due to the execution assistance the trailing wavefronts receives. Secondly, in an actual hardware implementation, due to artifacts in the coherence protocol, the caching mechanism, etc, there will be apparent, but false, races. Also, depending on the applications, some true races can play out differently without affecting the state of execution. Faithfully enforcing the same outcome for every one of those races may be unnecessary.

The actual implementation of a strategy depends on the coherence protocol and consistency model of the underlying multiprocessor. In this work, we focus on sequential consistency and invalidation-based snoopy coherence protocol. However, the architectural support can be extended to handle other coherence and consistency substrates.

3.2.1 Tracking the races

We track access orders in a broad-brushed manner to minimize the complexity and performance impact on the computing wavefront. Since a race involves at least one write operation, it will generate coherence actions: invalidation and/or downgrade of a dirty line. For instance, if store S_1 to cache line X from thread T_i happens before store S_2 to the same cache line from thread T_j, as S_2 takes effect, T_j will request X to be transferred and invalidated from T_i's cache. This shows that S_1 comes earlier in time. Rather than recording the specific relationship between S_1 and S_2 only, we draw a logical time line to separate instructions finished up to now in the entire wavefront from those that are yet to finish and broadly group them into different *subepochs*.

More specifically, when a potential race is detected, a wavefront-wide subepoch transition request is triggered (Figure 5). Upon receiving the request, the processor records the number of committed instructions so far and transitions into the next subepoch. Note that subepoch transition is merely a background accounting activity to mark which instructions belong to which subepoch – such that no races happen between two instructions in the same subepoch. There is no stalling at all in the computing wavefront. Later in the verification wavefront, by making sure that subepochs are properly serialized, we guarantee that all races maintain their original outcome. There is an economy of representation in that a single subepoch takes very little to encode (using the number of committed instructions) and a single transition can represent multiple races. Also, it avoids the need to track particulars such as which specific instruction brought a cache line in. This independently conceived design shares the same philosophy with Strata [28].



```
T1, Subepoch i:
ldq r5, 80(r30)
addl r11, 1, r11
lda r3, 4(r3)
lda r6, 4(r6)
ldl r4, 0(r4)

T1, Subepoch i + 1:
lda r5, 128(r5)
stq r3, 64(r30)
stq r6, 72(r30)
stq r5, 80(r30)

T2, Subepoch i:
stl r21, 0(r15)
xor r21, r22, r21
stl r24, 0(r25)

T2, Subepoch i + 1:
stl r17, 0(r10)
stl r11, 0(r23)
ldq_u r31, 0(r30)
ldq r6, 72(r3)
ldq r9, 0(r9)
bis r31, r11, r16

T3, Subepoch i:
ldah r1, 8192(r29)
ldq r5, 80(r30)
lda r9, –21208(r9)
ldl r7, 0(r25)

T3, Subepoch i + 1:
ldq r3, 0(r9)
lda r22, –32(r22)
lda r1, –20176(r1)
ldl r22, 0(r22)
ldl r1, 0(r1)
addq r3, r16, r16
```

Figure 5. Races create wavefront-wide subepoch transition.

To find out if a race needs to trigger a subepoch transition, we need to determine if the two accesses (from two different processors) involved already belong to different subepochs. Conceptually, this is quite straightforward to infer: if we track the subepoch number (SEN) when a cache line is last accessed, we can directly compare that to the requester's current SEN. In a real implementation, a single-bit tracking in the cache seems sufficient: whether the cache line has been accessed in the current subepoch. However, there are a few subtle points worth mentioning.

First, the concept of "current" is imprecise because events such as memory accesses or subepoch transitions are not atomic. For example, a memory request can be issued in one subepoch and the instruction can be committed in a later subepoch. Also, due to the delay in the propagation of the subepoch transition request, the current subepoch of a different processor may not be the same. Thus, we have to explicitly encode SEN in some structures. For instance, when a load issues and brings data from a remote dirty line, the data reply can not simply encode that the data has been touched in the current subepoch. Rather, it needs to encode what is the current SEN (say, 4). Upon receiving the reply, this SEN will be recorded in the requester's load queue. At the commit time of the load, we need to ensure that the SEN is at least 5. If that is not the case, a request to increment the SEN is broadcasted.

Second, to precisely track the necessity of subepoch transition, we need two bits to independently track whether the cache line has been written or read. For example, if a dirty line in processor P_i's cache is only read by P_i in the current subepoch, then a read request from a different processor does not need to trigger a subepoch transition, even though the line is dirty. Using only one access bit, we can not tell whether the access by P_i in the subepoch was read or write and have to conservatively increase the subepoch number. However, in our experiments, we found that using just one bit has minimum negative impact.

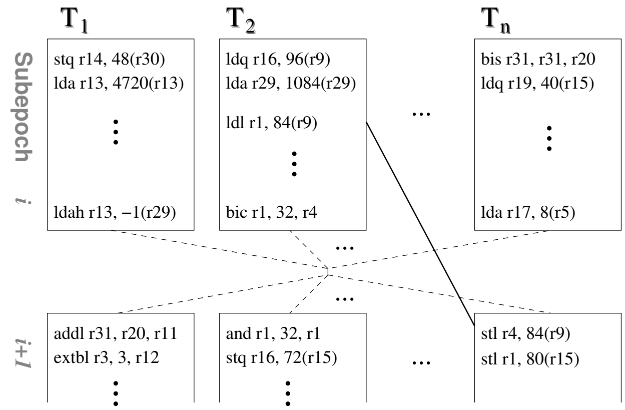
Third, when we are evicting a cache line, we may lose the access time information and may have to conservatively increase the SEN. For instance, consider the case where a cache line is read in the current subepoch and evicted and subsequently, a write request is issued from a different processor to the same cache line. In this case, since the cache line is not present anymore, the race can not be detected. Note that if the cache line has not been accessed in the current subepoch, then the eviction is not losing any information. A simple but conservative approach would be to initiate a subepoch transition when an eviction will lose information. To prevent unnecessary subepoch transitions, we add the following support that has little extra overhead. When we are evicting a dirty cache line whose PCB entry is still around, we can still maintain the timing information by writing the current SEN into the PCB. If the line does not have a PCB entry, we set an eviction bit for the cache set. Later on, when we receive an invalidation issued from a different processor and the address maps to the cache set with the eviction bit set, we treat it just as if we have the cache line that is being invalidated. That is, we trigger a subepoch transition. The eviction bits and the access bits are reset upon a subepoch transition.

Finally, we note that the epoch boundary is just a special case of subepoch boundary, where extra steps are taken. Any actions done at subepoch transitions are also done at epoch transitions.

3.2.2 Enforcing the order

The order tracking mechanism mentioned above passes on the number of instructions committed in each subepoch to the trailing verification wavefront, which uses the information to influence the execution order. There are a range of options to enforce this ordering. The only architectural support needed is that to freeze the commit stage.

Strict enforcement The first and the most straightforward approach would be to strictly enforce the ordering. That is, after committing all the instructions of the current subepoch, a processor has to wait for all other verification processors to finish the same subepoch before moving on. Note that with the hardware support for sequential consistency the processor can choose to fetch and execute instructions in the next subepoch, as long as they are not committed. The hardware guarantees the semantic effect follows the commit order through re-execution if necessary [61].



```
stq r14, 48(r30)
lda r13, 4720(r13)
ldah r13, −1(r29)

ldq r16, 96(r9)
lda r29, 1084(r29)
ldl r1, 84(r9)
bic r1, 32, r4

bis r31, r31, r20
ldq r19, 40(r15)
lda r17, 8(r5)

addl r31, r20, r11
extbl r3, 3, r12

and r1, 32, r1
stq r16, 72(r15)

stl r4, 84(r9)
stl r1, 80(r15)
```

Figure 6. The creation of a subepoch boundary implies not only the necessary ordering due to data race (solid line), but also false order dependences between the last instructions of one epoch and all the beginning instructions of the next (dashed lines).

However, strictly enforcing the ordering can lead to unnecessary waiting in the trailing wavefront which is inefficient. Recall that a race between two memory instructions can trigger a global subepoch transition which creates an array of extra false ordering constraints as shown in Figure 6. And these constraints are almost certain to cause some waiting in the trailing wavefront. Ironically, incurring unnecessary waiting due to false ordering constraints may be the only effect. This is because the race may be between two instructions far apart in time and their original order may have been naturally preserved in the verification wavefront, without the induced waiting.

Blindly speculating A second, more optimistic order-enforcement approach would be to initially disregard all subepoch transitions and assume either this does not lead to any order violation, or even if there is an order violation, the program outcome will not be affected. Of course, the assumption can be incorrect and we may find a difference in the program outcome. In that case, since the difference is perhaps more likely the result of not enforcing the correct memory order rather than the result of external noise mechanisms, we only roll back the verification wavefront and re-execute the epoch. This time, we strictly enforce the subepoch ordering. If the discrepancy persists, it will be due to errors and we then roll back both wavefronts.

Finally, we note that the discrepancy can be discovered early in the epoch: The computing wavefront supplied a branch prediction stream to the verification wavefront. In the error-free environment, the prediction should be perfect. Thus, if a misprediction occurs, it is very likely the result of actual control flow divergence due to different race outcomes. Therefore we can abort the execution early and roll back the verification threads to re-execute with strict enforcement.

Selective enforcement Finally, instead of the two extreme policies, we can employ a more middle-ground policy by selectively enforcing some ordering. The intuition behind the policy is that if the two memory operations in a race happened in close-by temporal proximity, their redundant copies in the verification wavefront are more likely to switch order than if they happened far apart in time in the computing wavefront. Therefore, if we can track timing, we can use this heuristic to proactively enforce ordering only for those subepoch transitions caused by "tight" races and only for the processors involved. Of course, we can still mis-speculate and find a discrepancy in program outcome. We follow the same process of first rolling back only the verification wavefront. If the discrepancy persists, we then roll back both wavefronts.

To track timing, we can use an additional *Recently-accessed* (R) bit per cache line, which is set on an access, and reset periodically by a timer. When a race is detected involving a cache line with the R bit set, it is regarded as a tight race. In a tight race, we want to ensure the ordering of the two subepochs, but only for the threads involved in the race. Specifically, if a race in the computing wavefront incremented the subepoch from i to $i+1$ and T_w and T_l are the winner and loser thread of the race respectively, in the verification wavefront, we make T_w issue a "release" signal only after the completion of subepoch i and T_l wait until all other threads have released subepoch i before starting $i+1$. Threads not involved in any tight race release as early as they can and do not wait to proceed to a subsequent subepoch.

4 Experimental Setup

To evaluate the proposed architecture, we simulate a 16-processor CMP. We use 8 as computing processors and 8 for verification. Our simulator is based on SimpleScalar [7] 3.0 toolset simulating the Alpha AXP ISA. Significant modifications are made including an event-queue infrastructure to faithfully track memory access timing and contention and faithful modeling of speculative memory instruction scheduling and replays [11]. The simulator models a 16-processor CMP with necessary structures of inter-processor communication. To enable simulation of parallel programs, system call support for thread creation, synchronization instructions ldl_l and stl_c (load-linked and store-conditional), and sense reversing barriers [12] have been implemented. Cache coherence is provided by a snoop based MESI protocol [12].

All the structures required for our design have also been faithfully modeled into the simulator. We have added rollback capability to our simulator so that we can faithfully model and evaluate the effects of rollbacks during execution due to memory order violation: if memory order is violated for the verification processors, the execution continues on with the (possibly) wrong value until it is detected at the end of the epoch when checkpoints are compared or through a branch misprediction. During this period of "faulty" execution, the simulated processor continues on fetching, executing, and committing instructions. After the violation is detected, verification processors are rolled back and the epoch is re-simulated.

We perform our experiments using 12 applications including those in the SPLASH-2 benchmark suit [57], a parallel genetic linkage analysis program *ilink* [14], and a parallel version of the traveling salesman problem [1] without optimization (*tspUO*) and with optimization to avoid false sharing (*tspO*). Though our system does not yet support commercial workloads, we note that *tspUO* has artificially high coherence traffic due to false sharing and is specifically introduced to stress-test the design. We follow the recommendation in [57] to scale down the L1 cache to mimic realistic cache miss rate. Table 1 provides the parameters for a single core of the CMP.

Fetch Queue Size	16 instructions
Widths	Fetch: 4 / Dispatch: 4 / Commit: 12
Branch Predictor	2048 entry BTB, bimodal, 2-level adaptive, 32 entries RAS, min. misprediction penalty: 12 cycles
Functional Units	3 ALU + 1 Mult (INT); 3 ALU + 1 Mult (FP)
Reg. File	128 Int, 128 FP
Issue Queue	32 Int, 32 FP
LSQ / ROB	64 entries/256 entries
L1 I/D cache	8 KB, 16B line, 2 way, 2 cycles
L2 Cache (shared)	2 MB, 32 way, 128B line, 20 cycle
TLB (I/D each)	128 entries, fully associative
Memory latency	200 cycles
PCB	32 entries per epoch, 8 epochs, 1 port
Epoch size	2048 instructions, 8 sub-epochs, or 32 unique PCB entries, whichever comes first
PCB bloom filter	257 entries, 8 bits per entry
Checkpoint logic	16 cycles (4 registers/cycle) for creation/loading

Table 1. Simulation parameters.

5 Experimental Analysis

PCB access optimizations The PCB is important to decouple the commit of verified data from the normal execution of the cores. However, associatively searching a PCB is costly both in terms of time and energy consumption. We described two optimizations in Section 3.1.2. Figure 7 shows their effectiveness in cutting down the number of associative searches. A PCB access may be needed due to a local or a remote request.

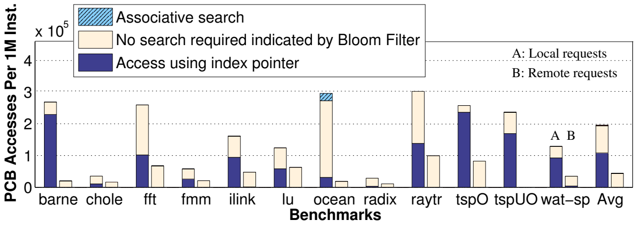


Figure 7. Breakdown of total PCB accesses per million instructions by search type. The first bar breaks down the PCB accesses for local requests. The second bar breaks down the access pattern for remote requests.

The first thing to note is that with the two techniques, only one application (*ocean*) still has a noticeable number of associative searches remaining. On average, less than 1% of the accesses need to perform an associative search. The remaining accesses either are stopped by the bloom filter (53.7%) or access the PCB directly via the pointer (45.4%). Overall, at about only 466 times per million committed instructions (geometric mean), actual associative searches are rare in absolute terms too.

Secondly, the two optimizations complement each other very well. Due to temporal and spatial locality, the index pointer is very effective when a processor accesses its own PCB. Even if the pointer is stale, it still indicates that there is no other version of the line in the PCB, so no search is needed. Out of all the local requests that would need to check the PCB, the pointer mechanism filters out 55.3% and the bloom filter filters out 43.7%. The index pointer is not as useful in a direct manner when it comes to handling remote requests (as evident from the second bar). Most coherence messages are to data not actually being shared and therefore will not match any cache line in other cores. However, the index pointer still helps in an indirect way: those lines being pointed to by a pointer do not set their presence bit in the bloom filter while their in-cache versions stay in the cache. This drastically reduces the number of bits set in the bloom filter and therefore cuts down the number of false hits. As a result, for the accesses that would otherwise search the PCB (cache miss for local requests and coherence requests that do not find a match in the cache), 98.8% are filtered. The remaining 1.2% are mostly false positives. Without this optimization, the filter becomes much more clogged and results in many more false positives. Except for one application (*ocean*), the number of false positives would increase by at least 21 times and as much as more than 800 times.

Memory access bandwidth impact As discussed earlier, dividing execution into epochs increases the PCB-to-L2 writeback traffic. Simply delaying the commit of PCB data to the L2 cache significantly mitigates the increase in traffic. We show the effect in Figure 8. The figure shows the traffic as a percentage of the available bandwidth between the L1 and L2. The two right hand bars for each application show the breakdown of bandwidth usage without and with delayed writeback from PCB. For reference, the leftmost bar shows the bandwidth consumption of running the computing wavefront alone in a non-redundant manner. In this case, writebacks happen when a dirty line is evicted.

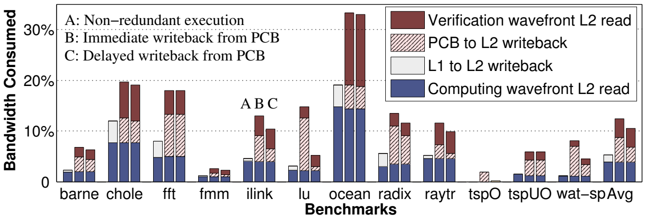


Figure 8. Core-to-L2 communication bandwidth consumption.

We can see that in all cases, writeback traffic only consumes a small portion of the available L2 bandwidth. Nevertheless, the *relative* increase due to epoch segmentation is dramatic in some applications. With a delayed committing, much of the unnecessary writeback is absorbed. For instance, in *lu* the reduction is 93%. On average, delaying writeback eliminates 37% of writeback traffic which in turn reduces the overall L2 traffic by 15% compared to the TLR scheme that writes back immediately after validation. Overall, as Figure 8 shows, the on-chip traffic to support a 16-thread TLR execution is about 96% higher than a non-redundant execution with 8 threads. Clearly, supporting TLR does not overtax the on-chip communication bandwidth. The impact on the off-chip bandwidth is even smaller. Because the two wavefronts share the same on-chip L2 cache, the off-chip bandwidth demand is only 7.9% higher than running a single wavefront.

Area and energy overhead Our TLR support requires some extra storage to perform buffering and bookkeeping. All such storage structures are off the critical path and do not require premium silicon real-estate close to the core pipeline. Furthermore, the space overhead is also modest. Table 2 lists the size of the structures. Note that for this calculation, we assume a more likely 32KB L1 cache. The power overhead of accessing these structures amounts to an insignificant 0.5% of the total power consumption.

Structure	PCB	Checkpoint buffer	Index pointer	Bloom filter	Checksum buffer	Branch info Q	**Total**
Size (Byte)	4096	4608	2048	257	128	128	11.3K

Table 2. Storage overhead per core assuming a typical 32KB L1.

Performance impact Redundancy necessarily reduces the amount of resources used for actual computation. In lock-stepped redundancy (*e.g.*, [44]) or TLR, half of the processors are not contributing to the computation *when redundancy is enabled*. This loss is even higher for products with triple-modular redundancy [23]. It is up to the user to decide whether such tradeoff between throughput and reliability is worth the while. As transient errors are likely to be rare relative to other events inside a microprocessor, the performance impact under fault-free situations is thus a very important metric.

We first measure this performance impact under the three different policies of memory access order enforcement as discussed earlier – *blindly speculating*, *strict enforcement*, and *selective enforcement*. For selective enforcement, we use a tightness window of 100 cycles, which catches a significant portion of the violating races without enforcing too many strict boundaries. Also, at boundaries that are enforced, threads stall at the commit stage, waiting for other threads to catch up. Stalling at commit is more efficient compared to stalling at dispatch as it utilizes the processor pipeline by continuing execution. However, stalling at commit increases the number of sequential consistency replays [61].

Figure 9 shows the execution speed of the three configurations all normalized to the system where all 16 processors are used for computation. The main source of the performance loss is redundancy itself, not our particular design. To highlight design-specific slowdowns, we also show an idealized TLR design approximated by running only the compute wavefront on half of the processors. In our suite of applications, the throughput loss in this idealized TLR configuration ranges from 27% to 49% with an average of 39%.

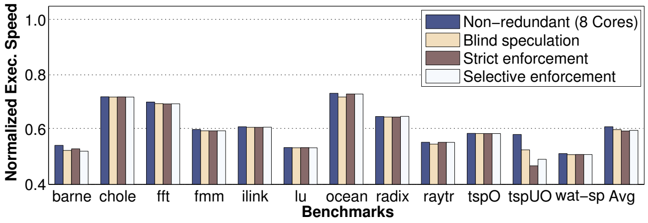


Figure 9. Performance impact of different configurations.

Compared to the idealized TLR, the additional performance impact is rather small. In all three cases, the average degradation is less than 3%. It is worth noting that in all cases, the direct overhead for checkpoint creation for the leading wavefront is negligible as we are only stalling the commit stage 16 cycles every epoch, which lasts on the order of 1000 cycles. In some applications, blindly speculating generates many rollbacks and slows down the execution noticeably. Figure 10 shows the percentage of the verification epochs that need to be rolled back due to incorrect race outcomes.

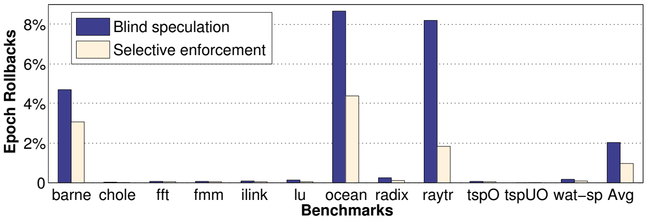


Figure 10. Percentage of epoch rollbacks due to relaxed order enforcement.

As we can see, selective enforcement reduces the number of rollbacks by 52% on average. Note that strict enforcement does not generate any memory order violation-induced rollbacks at all. From an overall performance perspective, strictly enforcing the order is a good option. However, this approach has its disadvantages. In the verification wavefront, we are waiting for the slowest thread within every subepoch, effectively placing far more fine-grain barriers than the other two options. As a result, the throughput of the verification wavefront is lower, and there is less "slack" to trade off for energy savings. Figure 11 shows this point quantitatively for the tested applications.

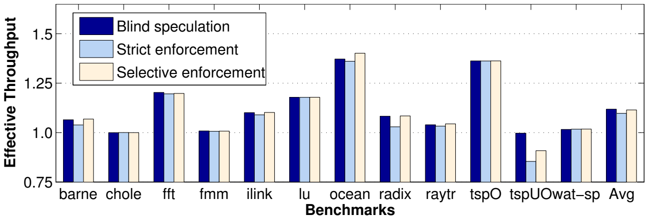


Figure 11. Effective IPC of verification processors for different fault-tolerant configurations.

The figure shows the normalized *effective* throughput of the verification wavefront, which is calculated as the throughput of the verification wavefront excluding the idle periods when there is no verification workload, and then normalized to the throughput of the computing wavefront.

We can see that with blind speculation, the verification wavefront has a 11.8% higher effective throughput than the computing wavefront, thanks to the execution assistance it receives from the latter. For our applications, some of that higher throughput is offset by extra work needed because of the rollbacks. For strict order enforcement, this advantage lowers somewhat to 9.7%. Clearly, this difference is application-dependent and for applications with more communications than those shown here, this gap will likely increase. Thus, it is still desirable to develop an effective strategy to minimize unnecessary stalls in the verification wavefront.

Controlling subepoch transitions We use the subepochs as an economic, broad-brushed race-tracking mechanism. Unnecessary transitions should be limited as they not only hurt the efficiency of the verification wavefront but also can induce frequent epoch changes which increases the overhead of the computing wavefront.

As described in Section 3.2, when a cache line is evicted, races involving the cache line can no longer be tracked by the baseline cache coherence mechanism. We can conservatively increase the subepoch number, but this will trigger too many unnecessary transitions as shown in Figure 12 by the right bar in each pair. Clearly, with the exception of *tspUO*, the great majority of subepoch transitions are due to cache evictions. The left bar shows the same breakdown when we use a single bit per cache set to remember eviction (and only trigger a transition when a coherence activity happens to a line with the eviction bit set). As we can see, the number of subepoch transitions drastically reduces: by 61% on average. With this simple mechanism, epoch transition due to running out of subepochs is extremely rare, except in *tspUO*. This is visually shown in Figure 13, which shows the total number of epoch transitions and the breakdown based on the reasons of the transitions. We can see that running out of PCB entries is the major reason whereas frequent subepoch transitions represent a much smaller portion of epoch transitions. Intuitively, a more sophisticated mechanism to limit subepoch transition will have diminishing returns, at least for these applications tested. In our experiments, on average, each epoch contains 866 committed instructions per thread. This means that with our design, one can easily buffer several thousands of unverified instructions per thread.

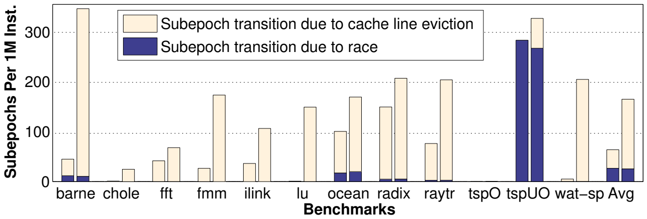


Figure 12. The number and breakdown of subepoch transitions per million committed instructions in each application. The left bar in each pair shows the number of subepochs with the per-set eviction bit. The right bar shows the result when an eviction simply triggers a subepoch transition.

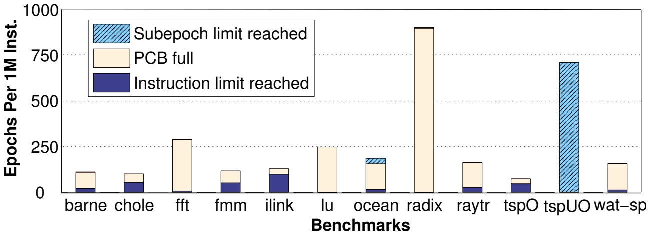


Figure 13. The total number of epoch transitions per million committed instructions and the breakdown of that number by the causes that trigger the transitions.

6 Related Work

There is a significant body of work exploring various forms of redundancy to enhance the system's tolerance to errors. In addition to the vast amount of work on circuit-level fine-grain redundancy discussed previously [8, 15, 19, 26, 32, 55], there are also numerous proposals for architecture-level redundancy either using dedicated, physically redundant hardware [3, 9, 44], or replicating instructions to use the same hardware or another thread context [4, 17, 27, 31, 35–37, 39, 44, 45, 53]. Such instruction replication can be done at the granularity of individual instructions [31, 36], or at the thread level, duplicating the thread either entirely or partially [4, 16, 17, 27, 37, 39, 44, 53]. Much of this prior work of architecture-level redundancy focuses on a single thread and typically relies on replicating results of a load instruction, and thus does not independently verify the correctness of the memory subsystem logic. In shared-memory parallel workloads, this leaves the cache coherence and memory consistency logic also outside the protection of redundancy. We described a design that forces redundant uses of the logic in the memory subsystem.

Without enforcing lock-stepping, redundantly performing the same memory accesses can produce different results due to timing non-determinism in parallel execution. Smolens et al. address this eventuality by resorting to synchronized requests in Reunion [47]. When discrepancy is detected, the redundant processor pair roll back to a known correct state, single-step until the first memory operation, and issue a synchronized request. We use a different approach: rather than relying on single-stepping to ensure deterministic results, we only affect execution timing by stalling the commit stage to avoid non-determinism.

Another body of work focuses on fault tolerance in the multiprocessor domain [5, 6, 22, 34, 38, 48–51, 58]. However, this work is mostly concerned with creating efficient or globally-consistent checkpoints. Error detection is typically not the concern. While errors in storage elements may be relatively easy to detect with error-correcting codes, errors in logic are much harder to detect and will quickly become non-negligible [42]. The redundant execution we used in this paper fills the gap of error detection. Another common assumption in this work is that verification is immediate because of using mechanisms like lock-step redundant execution [5, 49], error detection code, internal control check [38], execution validation routines [22], and so forth. In these cases, the two redundant threads are highly synchronized and will not experience different race outcomes. Our design allows highly-decoupled redundant threads.

Another focus of this body of work is on fast and efficient recovery mechanisms. In [38], rollback points are defined by the application program while others use more application-transparent check-pointing and recovery techniques. [21] proposes using a recovery cache with the PDP-11 processor to facilitate rapid recoveries. [58] augments the copy-back update policy with a normal cache, while [22, 51] use dedicated hardware buffers to store unverified memory updates similar to our PCB. By virtue of buffering unverified results, our system naturally supports fast recovery. The key goals of our design are efficiency and non-intrusiveness. The simple and yet effective use of index pointers and the unique way of applying the bloom filter are novel contributions compared to these buffers proposed.

Finally, a body of work addresses the issue of tracking the relative order of memory accesses for debugging [28, 59, 60]. The emphasis of order tracking for debugging and redundant execution for fault tolerance has subtle but important differences. While for debugging purposes it is necessary to be able to comprehensively and exactly reproduce an execution order, for our purpose, besides guaranteeing forward progress, we only need to minimize the chance of divergent execution due to different outcomes of memory access races. Also, while debugging support can afford expensive offline inference to reconstruct the exact execution order, we need a simple online enforcement support. As a result of the differences, our order tracking and enforcement mechanism uses a simple representation that matches well with our epoch-based computing/verification flow.

7 Conclusions and Future Work

Aggressive device dimension scaling is widely expected to bring significant challenges in maintaining system integrity in future technology generations. High-level compensation techniques are therefore becoming increasingly indispensable to offer end-users the required level of dependability. Such support needs to be flexible, supporting on-demand enhancement of dependability, and non-intrusive in terms of the impact to the design complexity, the circuit critical paths, and the performance of the applications. Thread-level redundancy (TLR) offers a very attractive paradigm as it can offer comprehensive protection and, with the ever more prevalent multicore architecture, it can be implemented non-intrusively, without affecting timing critical paths. Even though there is an extensive literature exploring TLR issues in the realm of single-threaded applications, efficient architectural support for comprehensive redundancy of multithreaded, shared memory workloads remains under-explored.

Our design uses simple and effective solutions to buffer memory state and to track and enforce memory races to deal with the non-determinism in multithreaded execution. All hardware supports are off the critical path, and can be disabled easily and cleanly. The core microarchitecture is redundancy-agnostic. Experimental analysis has shown that the novel design of PCB for memory state buffering and the optimization techniques to cut down expensive associative searches are very effective. The memory access order tracking and enforcement logic uses simple architectural support and yet is efficient in the common case, allowing the trailing verification threads to execute without excessive waiting. Overall, compared to an idealized TLR implementation, our TLR support only introduces a small 2.5% performance degradation.

In the future, we plan to study the extension of our architectural support and optimization techniques to other multiprocessor design points (such as hierarchical multiprocessor with multiple CMP chips), different coherence mechanisms, and consistency models.

Acknowledgments

We would like to sincerely thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] C. Amza *et al.*. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, **Vol. 29**(2):18–28, Feb. 1996.

[2] H. Ando *et al.*. A 1.3-GHz Fifth-Generation SPARC64 Microprocessor. *IEEE J. Solid-State Circuits*, **Vol. 38**(11):1896–1905, Nov. 2003.

[3] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Int. Symp. Microarchitecture*. Nov. 1999.

[4] D. Bernick *et al.*. NonStop®Advanced Architecture. In *Int. Conf. Dependable Systems and Networks*. Jun.–Jul. 2005.

[5] P. Bernstein. Sequoia: A Fault-tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, **Vol. 21**(2):37–45, Feb. 1988.

[6] A. Bondavalli, S. Chiaradonna, and F. Giandomenico. Efficient Fault Tolerance: An Approach to Deal with Transient Faults in Multiprocessor Architectures. In *International Conference on Parallel and Distributed Systems*. Dec. 1994.

[7] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, Jun. 1997.

[8] D. Chardonnereau *et al.*. Fault Tolerant 32-bit RISC Processor: Implementation and Radiation test Results. In *Single-Event Effects Symposium*. Apr. 2002.

[9] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In *Int. Symp. Microarchitecture*. Dec. 2000.

[10] Cisco Systems. Cisco 12000 Single Event Upset Failures Overview and Work Around Summary, 2003. http://www.cisco.com/

`warp/public/770/fn25994.shtml`.

[11] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Sep. 2000.

[12] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann, 1999.

[13] T. Dell. *A While Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory*. IBM Microelectronics Division, Nov. 1997.

[14] S. Dwarkadas *et al.*. Parallelization of General Linkage Analysis Problems. *Human Heredity*, **Vol. 44**:127–141, 1994.

[15] B. Gill *et al.*. An Efficient BICS Design for SEUs Detection and Correction in Semiconductor Memories. In *Design, Automation and Test in Europe*. Mar. 2005.

[16] B. Gold *et al.*. TRUSS: A Reliable, Scalable Server Architecture. *IEEE Micro*, **Vol. 25**(6):51–59, Nov./Dec. 2005.

[17] M. Gomaa *et al.*. Transient-Fault Recovery for Chip Multiprocessors. In *Int. Symp. Computer Architecture*. Jun. 2003.

[18] S. Hareland *et al.*. Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes. In *IEEE Symp. VLSI Technology*. Jun. 2001.

[19] P. Hazucha *et al.*. Measurements and Analysis of SER-Tolerant Latch in a 90-nm Dual-Vt CMOS Process. *IEEE J. Solid-State Circuits*, **Vol. 39**(9):1536–1543, Sep. 2004.

[20] R. Kumar *et al.*. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Int. Symp. Microarchitecture*. Dec. 2003.

[21] P. Lee, N. Ghani, and K. Heron. A Recovery Cache for the PDP-11. *IEEE Transactions on Computers*, **Vol. 29**(6):546–549, Jun. 1980.

[22] Y. Lee and K. Shin. Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks. *IEEE Transactions on Computers*, **Vol. 33**(2):113–124, Feb. 1984.

[23] L. Longden *et al.*. Designing A Single Board Computer For Space Using The Most Advanced Processor and Mitigation Technologies. In *European Space Components Conference, ESCCON 2002*. Sep. 2002.

[24] D. Lyons. Sun Screen. *Forbes*, Nov. 2000. `http://www.forbes.com/global/2000/1113/0323026a.html`.

[25] D. Mavis and P. Eaton. Soft Error Rate Mitigation Techniques for Modern Microcircuits. In *Int. Reliability Physics Symp.*. Apr. 2002.

[26] S. Mitra *et al.*. Robust System Design with Build-in Soft-Error Resilience. *IEEE Computer*, **Vol. 38**(2):43–52, 2005.

[27] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Int. Symp. Computer Architecture*. May 2002.

[28] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Int. Conf. Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[29] J. Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms form Unreliable Components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pp. 43–98. Princeton University Press, 1956.

[30] M. Nicolaidis. Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies. In *IEEE VLSI Test Symp.*. Apr. 1999.

[31] A. Parashar *et al.*. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Int. Symp. Computer Architecture*. Jun. 2004.

[32] J. Patel and L. Fung. Concurrent Error Detection in Multiply and Divide Arrays. *IEEE Transactions on Computers*, **Vol. 32**:417–422, 1983.

[33] W. Peterson. On Checking an Adder. *IBM Journal of Research and Development*, **Vol. 2**(2):166–168, Apr. 1958.

[34] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Int. Symp. Computer Architecture*. May 2002.

[35] M. Rashid *et al.*. Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance. In *Int. Conf. Parallel Architectures and Compilation Techniques*. Sep. 2005.

[36] J. Ray *et al.*. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Int. Symp. Microarchitecture*. Dec. 2001.

[37] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Int. Symp. Computer Architecture*. Jun. 2000.

[38] J. Rohr. STAREX Self-Repair Routines: Software Recovery in the JPL-STAR Computer. In *Int. Symp. Fault-Tolerant Computing*. 1973.

[39] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Int. Symp. Fault-Tolerant Computing*. Jun. 1999.

[40] S. Rusu *et al.*. A 1.5-Ghz 130-nm Itanium$^{\circledR}$ 2 Processor With 6-MB On-die L3 Cache. *IEEE J. Solid-State Circuits*, **Vol. 38**(11):1887–1895, Nov. 2003.

[41] N. Seifert *et al.*. Historical Trend in Alpha-Particle induced Soft Error Rates of the Alpha$^{\text{TM}}$Microprocessor. In *Int. Reliability Physics Symp.*. Apr. 2001.

[42] P. Shivakuma *et al.*. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Int. Conf. Dependable Systems and Networks*. Jun. 2002.

[43] B. Sinharoy *et al.*. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, **Vol. 49**(4/5):505–521, Sep. 2005.

[44] T. Slegel *et al.*. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, **Vol. 19**(2):12–23, Mar./Apr. 1999.

[45] J. Smolens *et al.*. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitecture. In *Int. Symp. Microarchitecture*. Nov. 2004.

[46] J. Smolens *et al.*. Fingerprinting: Bounding the Soft-Error Detection Latency and Bandwidth. In *Int. Conf. Architectural Support for Programming Languages and Operating Systems*. Oct. 2004.

[47] J. Smolens *et al.*. Reunion: Complexity-Effective Multicore Redundancy. In *Int. Symp. Microarchitecture*. Dec. 2006.

[48] D. Sorin *et al.*. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Int. Symp. Computer Architecture*. May 2002.

[49] L. Spainhower *et al.*. IBM's ES/9000 Model 982's fault-tolerant design for consolidation. *IEEE Micro*, **Vol. 14**(3):48–59, Feb. 1994.

[50] Y. Tamir and T. Frazier. Application-transparent Process-level Error Recovery for Multicomputers. In *Hawaii Int. Conf. System Sciences*. Jan. 1989.

[51] Y. Tamir, M. Tremblay, and D. Rennels. The Implementation and Application of Micro Rollback in Fault-Tolerant VLSI Systems. In *Int. Symp. Fault-Tolerant Computing*. Jun. 1988.

[52] J. Tendler *et al.*. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, **Vol. 46**(1):5–25, Jan. 2002.

[53] T. Vijaykumar *et al.*. Transient-Fault Recovery via Simultaneous Multithreading. In *Int. Symp. Computer Architecture*. May 2002.

[54] N. Wang *et al.*. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *Int. Conf. Dependable Systems and Networks*. Jun. 2004.

[55] J. Watterson and J. Hallenbeck. Modulo 3 Residue Checker: New Results on Performance and Cost. *IEEE Transactions on Computers*, **Vol. 37**(5):608–612, 1988.

[56] C. Weaver *et al.*. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Int. Symp. Computer Architecture*. Jun. 2004.

[57] S. Woo *et al.*. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Int. Symp. Computer Architecture*. Jun. 1995.

[58] K. Wu, W. Fuchs, and J. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Trans. Parallel and Distributed Systems*, **Vol. 1**(2):231–240, Apr. 1990.

[59] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Int. Symp. Computer Architecture*. Jun. 2003.

[60] M. Xu, R. Bodik, and M. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Int. Conf. Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[61] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, **Vol. 16**(2):28–40, Apr. 1996.

[62] J. Ziegler and W. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, **Vol. 206**(16):776–788, Nov. 1979.