Eric Hill
Pranay Koka

CS838 Final Report

Introduction

For our CS838 course project, we chose to look at availability solutions in CMPs. More specifically, we investigated portions of two recently proposed availability schemes, SafetyNet, and ReVive. The goal of this project was to characterize how performance would be affected if either of these two schemes were implemented. This performance characterization was done by contrasting the storage overheads created by SafetyNet with the bandwidth constraints imposed by ReVive. We focus primarily on the memory hierarchy in a CMP because that is the source of the majority of the overheads associated with availability. Both of the schemes we explore in this project focus on backward error recovery. Backward Error Recovery (BER) schemes periodically checkpoint system state, and rollback to a previously validated checkpoint upon fault detection [1]. This is in contrast to forward error recovery schemes, which rely on redundant hardware to detect system errors. The experiments conducted in this study all focus on how checkpoint storage affects both levels of caches in a CMP.

Motivation

Our main motivation for this project was that we view RAS (Reliability, Availability, and Serviceability) as an important feature of business class servers. For a large company, server downtime is equivalent to lost money. The ability to recover from hardware faults is a valuable feature for these servers. We believe that future business-class servers will be made up of CMPs, so we thought it would be worthwhile to study previously proposed SMP availability schemes in the context of a CMP.

BER Design Parameters

Checkpoint Consistency

Checkpoint consistency refers to when different components in an MP system decide to checkpoint their state. The simplest form of consistency is global, which is when every node in the system checkpoints its state at the same point in physical time. In a system which uses global consistency, all components must synchronize before checkpointing their state. In order to synchronize in physical time, a global skew less clock must be distributed throughout the system. This is relatively difficult thing to do in large systems. Also, after synchronization, all outstanding transactions must complete before the global checkpoint is taken [2]. These implementation details can potentially limit the performance of a system using this type of consistency.

A looser form of checkpoint consistency is coordinated local consistency. Components in systems using this form of consistency all checkpoint their state at the same point in logical time. A global logical clock must be distributed throughout such systems. System events such as coherence transactions can be used as a base of logical time, or a logical time base can be derived from a physical clock [3]. The latter option is used for our experiments.

The most unorganized form of checkpoint consistency is uncoordinated local consistency. Components in this type of system checkpoint their state independently of all other components. These systems also have the advantage of simplicity, but they have the problem of cascading rollbacks.

Recording of Checkpoint State

There are several different ways that system state can be checkpointed. The simplest method of checkpointing is flushing of data. The obvious drawback to doing this is that flushing consumes bandwidth that could be have been allocated to do useful work. Another alternative is incremental logging. Instead of flushing all checkpointed state, changes to the cache state can be logged as they occur. Logging allows the checkpointing overhead to be spread out over the entire checkpoint interval.

Location of Checkpointed State

For our purposes, there are only two options for where checkpointed state can reside, on chip or off-chip. The advantage of keeping the checkpointed system state on chip is that there is no overhead accessing the state during checkpointing or recovery. The primary drawback is that the checkpointed system state consumes on chip area that could have been allocated to other resources. Conversely, there is no storage overhead associated with storing checkpointed state off-chip, but there is overhead accessing the state during checkpointing and recovery.

SafetyNet

The first availability scheme we looked at, SafetyNet, was originally proposed by Sorin et al [3]. This scheme uses coordinated local checkpointing, incremental logging of data, and stores checkpointed state on chip in checkpoint log buffers (CLBs). In order to use coordinated local checkpointing, SafetyNet distributes a global logical clock derived from a physical clock throughout the system. This distributed clock also has a small skew. Sorin et al [3] argue that this skew is not a problem as long as it is smaller than the minimum transfer time through the interconnection network. As long as the skew is less than the minimum transfer latency, the situation where a message leaves one node at time n, and arrives as another node at time n – 1 is guaranteed not to occur [3].

ReVive

The second availability scheme we looked at, Revive, was originally proposed by Prvulovic et al [2]. This scheme uses global checkpointing, flushing of data, and stores checkpointed L2 cache state off chip in memory. We initially thought that ReVive would be a good idea for a single chip CMP system, since skew less clock distribution and synchronization should be less expensive in a CMP.

Simulator

A Simics-based CMP simulator provided by the Multifacet group was used to run experiments measuring storage and bandwidth. The means by which I/O statistics were collected is described later in this report. Table 1 below summarizes our simulation parameters.

L1 Cache Size	64 KB
L2 Cache Size	16 MB, 4-way set associative
Processor model	4-way OOO superscalar
Checkpoint Interval	100,000 cycles
Simulation Length	25 transactions

Figure 1. Simulation Parameters

Experiments

In order to collect data for this project, we added various event counters to the ruby simulator. A detailed description of the counters we used and their significance is included in the next section. We also modified the opal simulator to stop every checkpoint interval (which we set to 100,000 cycles), and return information about the counters placed in ruby. In order to estimate the number of cache blocks that need to be flushed for the ReVive scheme, we simply quiesce the shared L2 caches each checkpoint interval, and count the blocks in modified state. For a SafetyNet system, the period of the checkpoint clock can be considered to be 100,000 cycles, which a checkpoint being taken on each rising edge.

Implementation of SafetyNet Counters

L2 Counters

The SafetyNet protocol described in Sorin et al [3] logs cache blocks on the first update-action per checkpoint interval. An update action is best described as a transfer of coherence ownership. We identify 3 different kinds of update-actions, a GETX request coming from a local L1 cache, a forwarded GETX request from a remote CMP, and a replacement of a cache block in modified state. We denote the L1 GETX case as an *intrachip_gain* event, because ownership of the block can only be obtained by through a local store. Additional L1 GETX requests to an L2 block already in modified state are also logged as *intrachip_gain* events. Writebacks of L1 blocks in modified state to the L2 are also counted as *intrachip_gain* events. The case of a forwarded GETX request is denoted as an *interchip_loss* event, since the ownership of the block is lost due a message from a remote CMP. The case of a modified block being replaced is denoted as *intrachip_loss,* because the block was replaced due to a local reference.

L1 Counters

We identified 5 different events that can trigger update actions at the L1 cache. These events are stores from a processor, a writeback of a modified block, an invalidation message (for a modified block) from the L2, and the replacement of a modified block. Each time one of these events occurs, we increment our L1 counter if the event is the first update-action for that block for the current checkpoint interval.

Estimation of CLB Size

In order to come up with actual CLB sizes, we used the following formula:

CLB size = (*CLB entries created in one checkpoint interval*) * 5 * 72

We then multiply the number of CLB entries created by 5 because we assumed that in addition to the current checkpoint, there are 4 previous checkpoints awaiting validation. We assumed that each CLB entry is 72 bytes, with 64 bytes being for the cache line itself, and 8 being for tag and coherence state, as mentioned in [3].

ReVive Flushing Estimation

As was mentioned above, in order to estimate the cache blocks that must be flushed when a ReVive-like scheme is used, we simply quiesce the L2 caches counting blocks that are in modified state. Because ReVive-like schemes use global checkpoint consistency, a system using ReVive needs to synchronize before taking a checkpoint, waiting for all outstanding transactions to complete. This is referred to as a two-phase commit operation in [2]. We did not consider the synchronization latency in our study, but in addition to counting modified blocks in the L2 cache we also count blocks in transient states that would eventually end up in modified state. Interesting future work would be to model how many cycles this synchronization latency would actually take.

Design Assumptions

In this section we present our system parameters and design assumptions. For our studies we used an on-chip two-level cache hierarchy with inclusion. Each core has a private 64 KB writeback L1 cache, with all L1's on a die sharing a common L2 cache. In this project we assume that only the processor state and on-chip cache state are checkpointed. We focus on the checkpointing of the cache state because the amount of processor state that must be checkpointed is negligible when sizes are compared. A 2-level hierarchy with writeback L1 caches requires checkpointing at both the levels, irrespective of whether inclusion or exclusion is followed.

Results presented in the next section use the above described memory system. Checkpointing in hierarchies using write-through caches however depends on whether inclusion or exclusion is used. With inclusion, checkpointing becomes a lot simpler in that L1 caches need not be checkpointed. This is because any store to L1 is written to L2 and L2 has all blocks that L1 has. This is not the case when exclusion is used. In exclusive cache hierarchies both L1 and the L2 needs to be checkpointed. This because modified blocks in the L2 can be replaced (and hence the block is discarded from L2) while it is still valid and modified in L1. In this case, checkpointing the L2 alone will not checkpoint that modified block.

Results

In this section, the results of our experiments are discussed. One major point that should be noted about the graphs in this section is that they are normalized to transactions. When actual CLB sizes are calculated, the number of CLB entries created per ***interval*** is used. We normalized the graphs in this section to transactions in order to illustrate how the storage overhead and flushing costs vary for identical amounts of work completed by each configuration.

SafetyNet storage overhead

Figures 1 and 2 show how the L2 storage overhead varies as more cores are placed on a die.

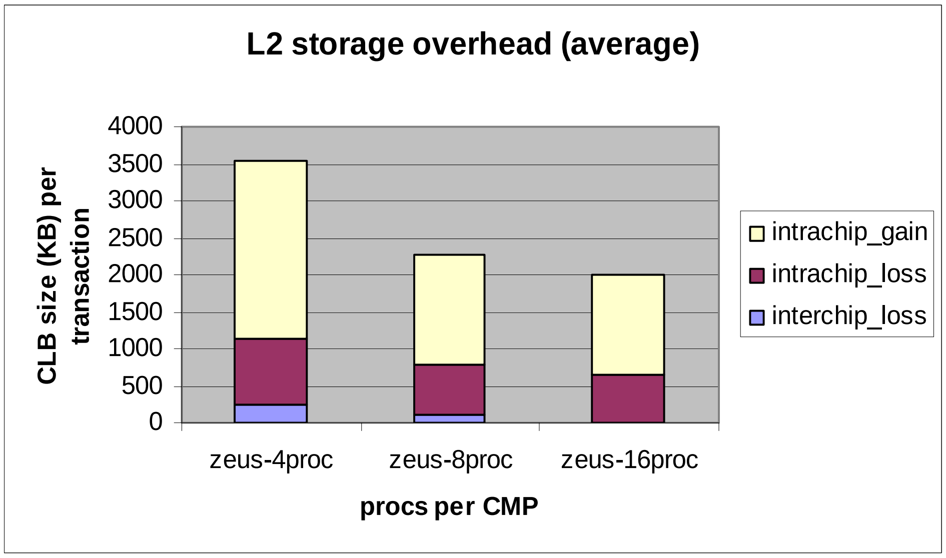

Figure 1. L2 storage overhead for zeus server workload

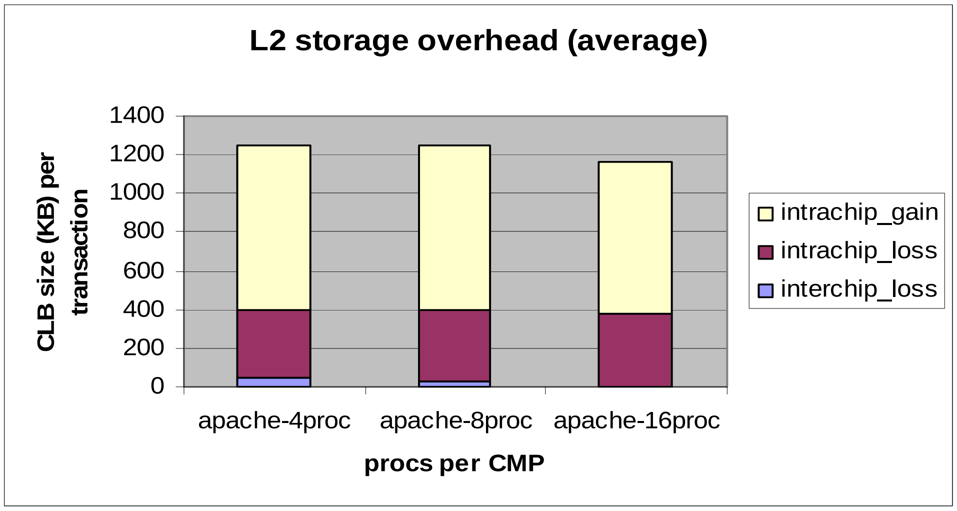

Figure 2. L2 storage overhead for apache server workload

Of the components that contribute to CLB size, interchip loss is the least significant and diminishes to zero as more processors are put on a chip. This is an intuitive result. In an MP system, as more and more processors are placed on single chips, more CLB entries will be created due to intrachip events. Though we do not have the data, we suspect that interchip_loss events are a much larger portion of CLB overhead for the 2-processor CMP and SMP cases. From figures 1 and 2 it is clear that as CMPs have more processing cores, the storage overhead per transaction decreases only slightly for the apache workload, but more dramatically for the zeus workload. While this may seem like a positive result initially, it is important to note that storage overheads are calculated per ***interval***, not ***transaction***, so the 16 processor CMP actually has a much higher overhead since it does 25 transactions in significantly fewer checkpoint intervals than the 4 or 8-processor CMPs. So the absolute cost still increases as the number of processors continues to increase.

One important thing to note about the results in Figures 1 and 2 is that the average case for overhead is plotted. In a real system writes are bursty, so it is best to size the CLB to handle these cases rather than the average. Figure 3 shows the worst case overhead calculated for both the apache and zeus workloads.

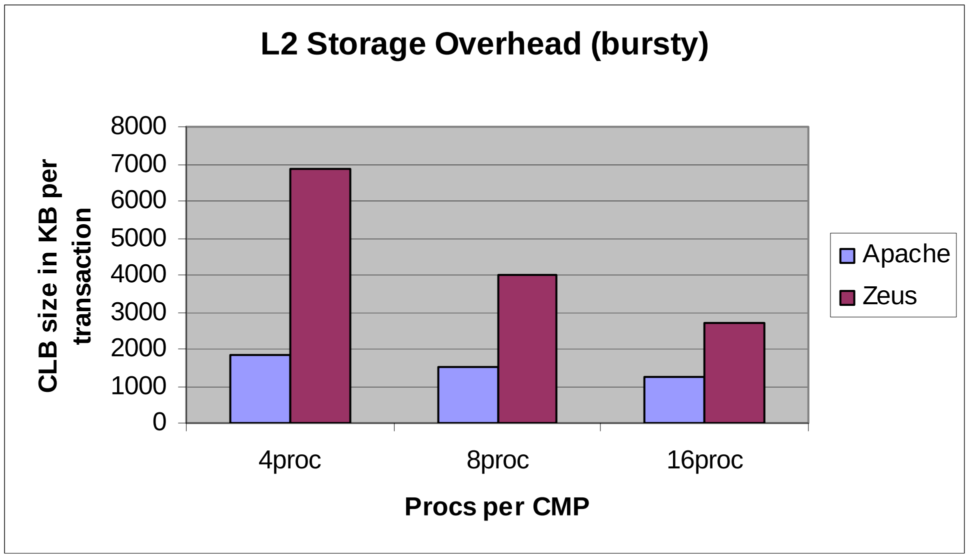

Figure 3. Worst-case storage overhead

Again, it should be noted that CLB size is calculated by taking the number of CLB entries created per interval, not per transaction. The actual CLB size for the worst case, a single CMP with 16 processors running the zeus workload, is 4.5 MB. In order to roughly estimate how 4.5 MB of L2 cache overhead would affect performance, we resized the L2 cache to 12 MB and ran the zeus workload one a single 16 processor CMP. We reduced the cache size by removing a way of associativity. We found that there is an 11.75% reduction in IPC with the smaller cache size. IPC is not the most optimal measure of performance in a multithreaded workload due to non-determinism, but we feel here that it roughly estimates the expected performance loss.

Figure 4 illustrates how the L1 storage overhead increases as more processing cores are placed on a die.

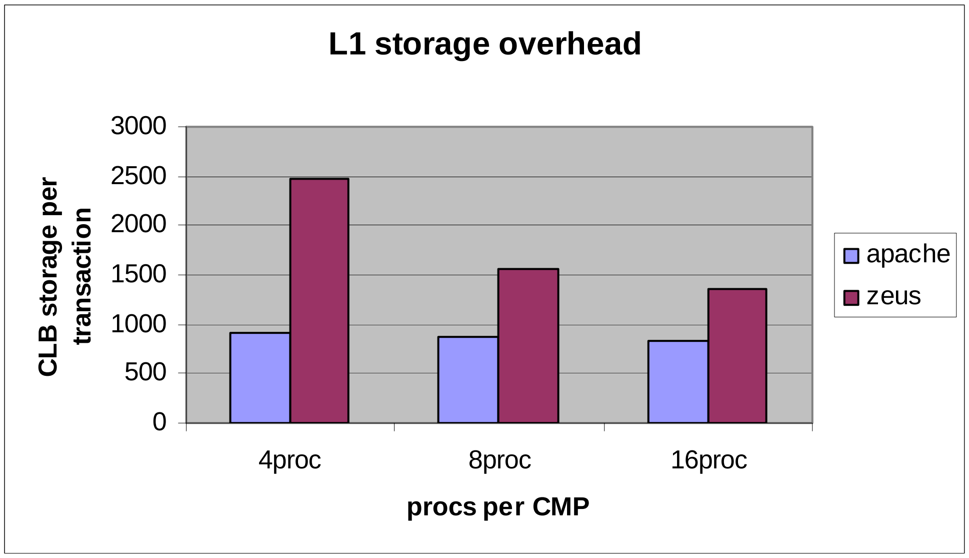

Figure 4. L1 storage overhead

Figure 4 shows that the L1 CLB storage overhead per transaction decreases for larger CMPs, but a decrease in CLB overhead per transaction does not directly translate into a decrease in the actual CLB size because varying amounts of work are done per checkpoint interval in each configuration. For the zeus workload, the storage overhead per transaction is actually larger than the aggregate L1 cache size for each case. This leads us to believe that using SafetyNet for L1 caches is a bad idea for write back caches. Using write-through caches would be a good workaround for this problem, since all logging could be done at the L2 level. Though this would alleviate the logging problem at the L1 level, though there would be much more logging at the L2 level. Interesting future work would be implementing studying SafetyNet storage overhead requirements in a memory hierarchy with write-through L1 caches and a shared L2.

Revive Bandwidth Overhead

In a system using ReVive for BER, all modified blocks are flushed to memory at the end of every checkpoint interval. Figure 5 shows how many dirty blocks (on average) need to be flushed from the L2 cache per transaction.

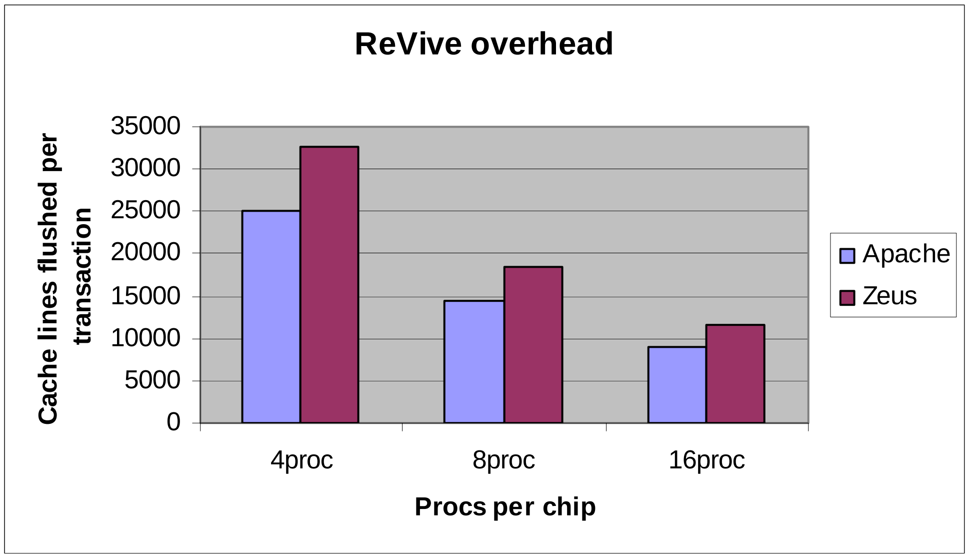

Figure 5. ReVive flushing overhead.

As more processors are added to a CMP, the number of cache lines flushed per transactions drops significantly. This indicates that the absolute number of lines per interval flushed does not directly depend on how many processors are present on a CMP, and should be relatively similar for all 3 configurations. Despite this fact, the amount of blocks flushed in all cases is large enough that we suspect that there would be significant performance degradation. Interesting future work would be to conduct a detailed study of how flushing impacts the interconnect bandwidth of the system. A major limitation of our ReVive analysis is that we only study how many blocks need to be flushed from the L2 cache. Another interesting question that we leave to future work is exploring strategies for also flushing the L1 caches.

Buffering I/O

In checkpoint based recovery schemes, data cannot be committed to the I/O device until the execution that generated the data is validated. The availability schemes proposed previously do not deal with this problem in depth. These schemes assume the I/O is buffered until the checkpoints are validated. The I/O commit problem has two components. One is the memory overhead incurred in buffering data. The other is the time criticality of I/O data. Some real time devices place strict constraints on the response times. Hence the data needs to be delivered to the device under strict time constraints. Hence the I/O commit problem is influenced by, to a great extent, the validation latency and the checkpoint interval.

In this section we show some data relating to buffering requirements needed for I/O in commercial servers. I/O in commercial servers is primarily from the network and disks. Table 2 shows the network and disk I/O rates of two commercial server benchmarks. We used TPC-B postgres database as one of our workloads. We ran TPC-B with 100 branches and 40 clients. The other workload was SPECweb9_ssl with apache web-server. We ran the benchmark with 60 clients and with 50% of the transactions being dynamic web transactions. For both the benchmarks we tuned the workload on a one processor Pentium 4 (GHz) processor to achieve a CPU utilization of 90-95%.

	Disk Writes	**Network Send**	**Network Recv**
TPCB	7098.6 KB/s	131.4 KB/s	1.4 KB/s
SPECweb99_ssl	32.7 KB/s	1325.6 KB/s	0.5 KB/s

Table 2. Disk and Network I/O rates

We think the disk reads are not a problem, as pages can be read speculatively using the un-validated data and squashed on a fault. In order to measure the data-write rate of the benchmark, we instrumented Linux 2.4.20 kernel to count data-writes. We measure the network writes and reads to estimate both the output and input commit problems. From table 2 we can see that buffering for network input commit is not a problem. The buffering requirement for disk and network write are more than that of the network input, but still are quiet manageable at least for a one-processor system. It would be interesting make a similar estimation on 16 or 32 processor CMP system.

Conclusions and Future Work

We acknowledge that there are several major limitations of this study. First, all of our simulation runs were only run for 25 transactions. Ideally we would have like to run for much longer, but we were bound by time constraints. Nevertheless, we believe that the data we collected in this study has value. From our experiments, we believe that using a SafetyNet solution for error recovery will have a smaller performance loss than using ReVive. The storage overheads incurred, however, would be significant. We are particularly concerned about the overhead of checkpointing the L1 caches since the storage need per transaction is larger than the aggregate L1 cache size in some cases. Also, despite the fact that the storage cost per performance decreases as more cores are added to a CMP, the absolute overhead (in terms of chip area) will continue to increase as more processors are added to a single chip. From our I/O study we also feel that the space needed to buffer I/O wouldn't be too much of a problem at least for a one processor system. As CMPs scale to more processors per chip, however, this buffering may become a problem. Further exploration of this issue would require would also require evaluation of different workloads. The question of whether or not CMP system with an availability solution could meet the I/O response times required by time critical devices, is also question that warrants future study.

References

[1] R. Ahmed, P. Frazier, and P. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. In *Proceedings of 20th International Symposium on Fault-Tolerant Computing Systems*, pages 82-88, Newcastle, June 1990.

[2] Milos Prvulovic, Zheng Zhang, and Josep Torellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111-122, May 2002.

[3] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123-134, May 2002.