A Mechanism to Verify Cache Coherence Transactions in Multicore Systems

Rance Rodrigues, Israel Koren and Sandip Kundu
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst MA 01003, USA.
Email: {rodrigues, koren, kundu} @ecs.umass.edu

***Abstract*—The functional correctness of shared memory applications executing on multicores and multiprocessor systems is supported by cache coherence protocols. The correct operation of these applications thus depends on the correctness of the cache coherence transactions. However, verifying the correctness of these transactions is not trivial since even simple coherence protocols have multiple states. Transitions among the states can fail due to aging of devices or single event upsets. In this paper we present a centralized mechanism for online verification of cache coherence transactions in snoopy bus multicore systems. We make use of an architecture that we previously proposed for opportunistic Dual Modular Redundancy (DMR). This architecture includes, in addition to the general-purpose cores, a diminutive core called the Sentry Core (SC) that is small and simple and thus, can be assumed to be fault-free. Like other cores, the SC has access to the shared bus and is aware of the cache coherence protocol. It monitors all bus transactions and by observing the current state of the cache line being addressed and the type of operation (e.g., read or write) it knows the expected next state for that cache line. Deviation from expected behavior will indicate a possibe error. Our preliminary experiments show that a significant fraction of the coherence transactions can be verified by our scheme.**

***Index Terms*—Online error detection, cache coherence, verification, centralized mechanism**

I. Introduction

The relentless push in technology scaling has led to smaller transistors and higher device densities but unfortunately, resulted in an increase in error rates [2], [3], [6]. The complexity of present day chips makes it almost impossible to detect all logic bugs pre-silicon [5]. Furthermore, devices are expected to experience errors during the operation in the field. Consequently, there have been a number of proposals for post silicon validation. Unfortunately, most of these schemes if applied online lead to significant performance penalties.

Multicore and many core systems rely on inter-core communication via shared memory. In such systems it is necessary to make sure that data consumed by all the cores is up to date. Cache coherence protocols help ensure this [20]. Functional correctness of shared memory systems thus depends on the correctness of the coherence hardware support. Ensuring correctness of the coherence hardware is difficult as even simple protocols can have multiple states [18]. The state space further increases when considering the state of a cache line shared across cores. Thus, there is need for an online mechanism to verify the operation of cache coherence transactions.

In this paper, we propose an online scheme to verify the operation of the cache coherence hardware in a snoopy bus multicore. We leverage an architecture that we proposed previously for opportunistic DMR in Chip Multi-Processors (CMPs) [8]. That architecture includes, in addition to the general-purpose cores, a small and simple core called the Sentry Core (SC) that is assumed to be fault-free. This assumption is akin to similar assumptions used in watchdog processors [7], [9] and the DIVA checker [1]. The SC has access to the shared bus, just like the other cores (see Figure 1). It monitors and logs all bus transactions, and is aware of the cache coherence protocol being implemented in the system. By observing the source and type of bus transaction, it can predict the expected next coherence state of that line for the requesting core and all other cores that share that line. Whenever the same line appears on the bus again, the SC can verify that it transitioned to the correct state. If not, an error is flagged. Our experiments using the SPLASH-2 [13] benchmarks suggest that a significant fraction of the transactions can be verified by the SC by simply monitoring the shared bus.

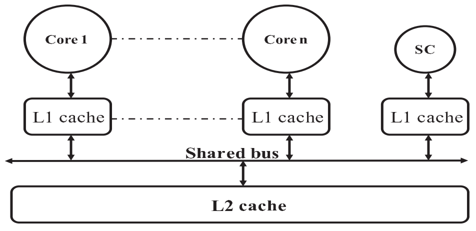

Fig. 1. A Sentry Core (SC) in a shared memory multicore.

The rest of the paper is organized as follows. In section II we provide a literature survey on the existing cache coherence verification mechanisms. In Section III we present details of the architecture to enable cache coherence verication. The experimental setup is discussed in Section IV followed by the results and conclusions in Sections V and VI, respectively.

II. Related work

We present in this section, a brief summary of the literature that closely relates to our proposal and point out the key differences.

In [11], Cantin *et al.* presented a variation of the DIVA checker [1] for cache coherence verification. Just like DIVA does for functional correctness of the cores, cache coherence transactions were verified using simpler logic. However, this scheme requires the use of a separate network for global verification of coherence states. In [12], Fernandez-Pascual *et al.* present a scheme for cache coherence verification in the presence of network failures. This scheme cannot be used to ensure correct transition of coherence states. A scheme to verify cache coherence with token coherence was proposed by Meixner *et al.* in [17]. The scheme requires implementation of logical timestamps, signature generation and comparison hardware. In [10], Borodin *et al.* present a distributed system to verify cache coherence. In their solution, each cache that participates in the coherence protocol is assigned a checker that verifies its operation, which enables local verification. Global verification is done by observing the shared bus. This scheme is closest to ours, but its overhead increases linearly with the number of cores in the CMP, unlike ours where a single SC services a number of cores. Furthermore, as will be shown later in the paper, this scheme may be too conservative. In [5], DeOrio *et al.* present an algorithm to verify cache coherence post-silicon. This algorithm, if implemented online, imposes a 26% performance penalty which is unacceptable. Verification of the cache coherence protocol itself was introduced by Zhang *et al.* in [18]. We next present our SC-based cache coherence verification scheme.

III. The proposed solution

We propose the use of the SC for verifying the coherence protocol in snooping bus multicores. General working of the system and a possible implementation of the system in real hardware are decribed next. In this paper, we assume the use of the MESI protocol [20], but our approach can be applied to any coherence protocol. We refer to the various MESI states as M-Modified, E-Exclusive, S-Shared and I-Invalid throughout this paper.

A. General working

The SC monitors all transactions on the shared bus and makes decisions about the correctness of the transactions. Three steps are involved in the process: (i) Transaction logging, (ii) Verification, and (iii) Retirement.

1) Transaction logging: This is the first step of the cache coherence verification mechanism. Whenever a cache line is requested due to a read/write miss, it is logged into the L1 cache of the SC. The hardware mapping of each cache access into the SC cache in described in the next sub-section. We assume that along with the address of the memory line being requested, its current coherence state in the sending core is also broadcast. The same assumption has been made by Borodin *et al.* [10]. The SC logs the address of the access, current state of the line and, depending on the transaction, the expected next state of the line. For a given cache line address, entries are maintained for each core in the system. When the line is shared among cores, the corresponding entries are updated, whenever such information is observed on the bus.

2) Transaction verification: After a request is logged, it is verified once the line appears on the bus again. There are two types of verifications that need to take place, i.e., (i) Local and (ii) Global. Local verification is conducted by computing the expected next state of the transaction for the same core. Whenever the same line appears on the bus, the SC can check if the line transitioned to the expected state. For example, if a core has a read miss and the line was not found in the L1 of any other core, its expected next state should be E (*Exclusive*) since it has exclusive access to the line. Global verification happens by making sure that the state of this line is consistent across cores. For example, a line existing in the S and M states in the L1 caches of two cores is an invalid situation that must be detected. This is done by the SC by comparing the state of the line in each core, that is logged in its own cache. Verification (local or global) happens whenever the line in question appears on the bus. There are two ways in which this happens. The first is when a core requests a line that is present in the cache of another core. In this case, the owner core will respond to the requesting core with a copy of the line. This information is broadcast on the bus along with the current coherence state of the line. Since this entry must have been logged earlier along with the current state in the logging stage, the SC now has access to the next state of that line. Comparing the current state to the state predicted by the SC enables local verification. If the line is shared by multiple cores, global verification is done by assessing the state of the line across cores. At this stage a new entry is created for the requesting core and the relevant entries are updated in the SC cache. The second verification opportunity arises when the cache of a core is full and lines need to be evicted to make space. If the line that is to be evicted is dirty (M state), a write to memory is initiated so that the main memory is kept up to date. When a write is initiated, the address of the line and its current state are broadcast on the bus and the SC then checks for local and global verification.

3) Entry retirement: If every line that was accessed is logged but never retired, the SC would need an unlimited cache size to log all the entries and the scheme would not be practical. However, logged cache lines are not needed for an unlimited period of time. Cache lines upon cache conflict have to be evicted from the L1 cache. If the line is in the dirty state, it is written to main memory. Since the cache line is dirty (M), no other core can have a copy of the line and once it has been evicted from the L1 cache, the corresponding entry in the SC L1 cache is retired. Sometimes, cache lines are in states other than M (S or E) and in this case, upon cache conflict, these lines will be overwritten (since the line is consistent with main memory in the S or E states and we do not care about lines in the I state). Whenever this happens, our scheme requires that the SC be notified via the shared bus. The entry is then retired from the SC cache. This event is expected to incur a small penalty, since it increases traffic on the bus. However, we have observed this penalty to be negligible in our experiments. Entries are also evicted from the SC cache when they are invalidated (I). This also implies that any transaction that appears on the bus with a state other than I, it must be logged in the SC cache, otherwise, an error has occured.

B. Mapping cache access transactions in hardware

We have discussed how the SC logs, verifies and retires transactions from its own cache. In this section we describe

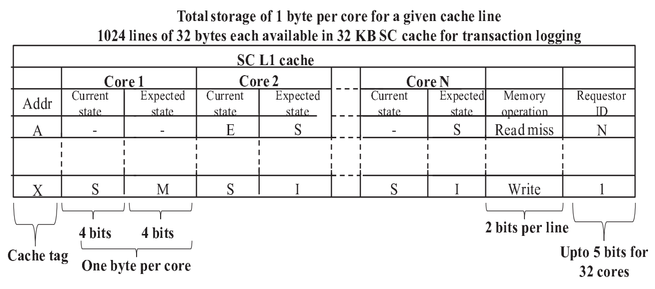

Fig. 2. Mapping cache transactions for each core in the SC cache.

how each transaction is mapped into the SC cache in hardware. In the considered system cache lines are assumed to be 32 bytes. The SC cache is assumed to be the same size as that of the general-purpose cores. We have assumed a 32KB L1 cache size and hence the total number of lines available is 1024. The SC cache addressing is done using the same address as that of the operation broadcast on the bus. Since we use the MESI protocol, we assume 4 bits each (13 total states along with transients) for current and expected next states of each core. This requires 1 byte per core. There is also need to log the current memory operation for each line and the requestor ID. Depending on these fields and the current state, the SC can compute the expected next state. Transactions that appear on the bus are either due to a read/write miss, memory push or invalidate. Hence, two bits are reserved for the memory operation and 5 bits for the requestor ID, which allows addressing up to 32 cores. The memory operation and requestor ID fields together occupy a byte, leaving the other 31 bytes to store records for up to 31 cores in the system. The SC cache is implemented as any general-purpose core cache, i.e., with tags, sets and offset. Tags and sets are computed using the address of the memory operation on the bus. Offset is computed using the requestor core ID. Figure 2 depicts the SC cache and its entries. Note that for lines exclusively held by a single core, just one entry (1 byte of the available 31) will be used and the rest will be wasted. Also if the number of cores in the system is less than 31, many entries are never used. Instead, if the SC cache was customized such that the number of bytes per line is equal to the number of cores in the multicore, not only would the SC cache be used more effeciently, there would be more entries to store additional cache transactions. However, this would complicate the design of the multicore. To avoid this, we assume that the line size in the SC cache is identical to that in the general-purpose core caches, i.e., 32 bytes.

C. Putting it all together

An example summarizing the above description is presented in Figure 3. For simplicity, the memory operation and requestor ID fields in the SC cache have not been shown, but appear in the text in the figure. All state updates are indicated in italic fonts in the caches and any state verifications are indicated by a star alongside the line state. In the example, two cores are considered. The contents of each core cache and SC cache are shown in stages A through E. In stage A, Core 1 has exclusive access to the line at address A. Its state

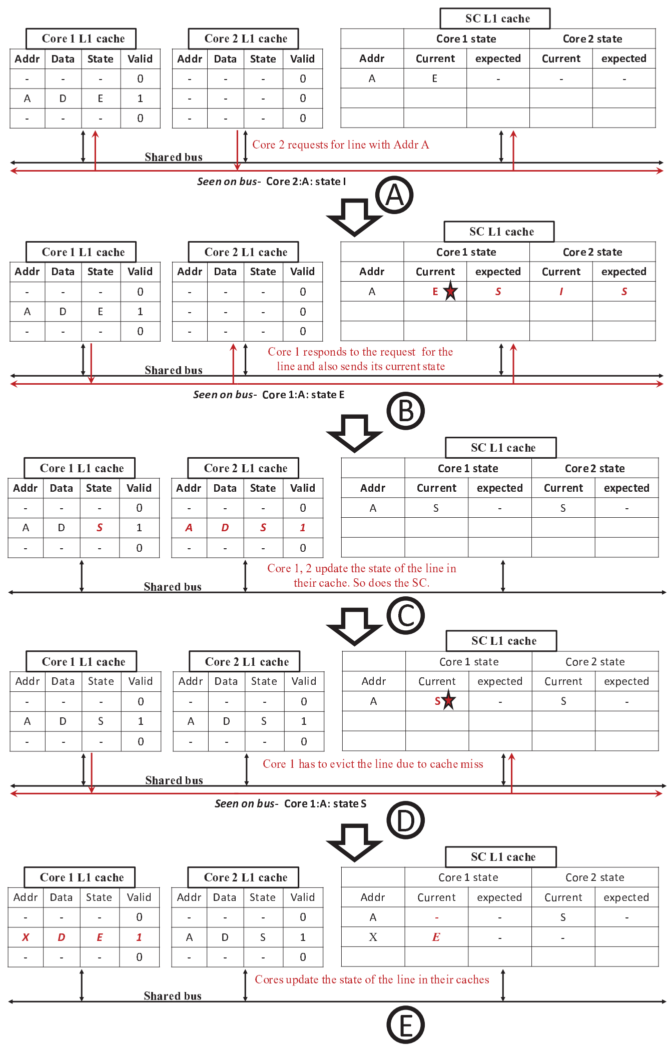
Core 1 L1 cache
Core 2 L1 cache
SC L1 cache
Addr
Data
State
Valid
Core 1 state
Core 2 state
Current
expected
Shared bus
Core 2 requests for line with Addr A
Seen on bus- Core 2:A: state I
A
Core 1 responds to the request for the line and also sends its current state
Seen on bus- Core 1:A: state E
B
Core 1, 2 update the state of the line in their cache. So does the SC.
C
Core 1 has to evict the line due to cache miss
Seen on bus- Core 1:A: state S
D
Cores update the state of the line in their caches
E

Fig. 3. Working example of transaction logging and retirement. The state verifications are indicated by a star alongside the state in the SC cache and any new state updates are indicated by italics font.

is recorded in the L1 of Core 1 as well as in that of the SC. Core 2 then requests a read for that line. This request is sent on the bus and seen by Core 1 and the SC. The SC logs this request and knows, based on the memory operation and requestor ID, what are the expected next states for both cores. The SC accordingly updates those fields for each core. In stage B, Core 1 responds to the request from Core 2 and broadcasts the state of the line in its cache. This helps the SC to verify the expected state for Core 1. In stage C, a static snapshot of the system with updated states is shown. In stage D, Core 1 has a cache miss and has to evict the line. This is observed on the bus and the SC can once again verify its operation. Stage E shows a static snapshot of the states in the system after the memory eviction for Core 1 is complete and the new line with address X having arrived.

D. Mathematical upper bound on the number of transactions that may be logged and verified

The SC cache is used to log and verify transactions. Hence, the size of the SC cache determines the upper bound on the

TABLE I
CHOSEN CORE PARAMETERS

Parameter	Value	Parameter	Value
Issue	6	INTREG	96
FPREG	80	INTISQ	36
FPISQ	24	Load/Store units	3
LSQ	32	ROB	128
L1(I/D)	32K	L2	2M
L1 associativity	8	L1 Linesize	32 bytes
L2 associativity	8	L2 Linesize	32 bytes
L1 hit latency	2 cycles	L1 miss latency	10 cycles
L2 hit latency	15 cycles	L2 miss latency	200 cycles
Freq (GHz)	2.4	Operation	Out of order

verification coverage that may be achieved. We now discuss the desired SC cache size that will allow all transactions to be verified. For simplicity, we assume a fully associative cache.

The minimum size of the SC cache required to log all transactions is $\sum linesValid$, where $linesValid$ is the number of valid L1 cache lines, where no two lines have the same address in memory. Note that for two cores sharing a line, only a single entry will be maintained in the SC cache (refer to Section III-B). The worst case arises, when every L1 cache line in the multicore is valid and none are shared. In that case, the minimum size of the SC cache is then $n * lines$ where $lines$ is the number of cache lines per L1 cache. In other words, the SC cache must be equal to n times the L1 cache size. It may be noted that this calculation was done using fully associative caches which is not always practical. Considering more realistic set-associative caches this minimum requirement on the cache size may be larger than that just calculated. However, as will be seen in the results, a key observation enables us to keep the required SC cache size realistic.

IV. Experimental setup

The shared memory multicore was simulated using the SESC simulator [4] which was modified considerably to enable cache coherence transaction verification via the SC. We used the SPLASH-2 workloads [13] for our experiments (*cholesky, barnes, fft, fmm, lu, ocean, radix* and *water*). Each core in our multicore represents an Intel Nehalem processor. The specifications of the core parameters that we have used are shown in Table I. We consider 8 cores in the multicore for all our simulations and we simulate the workloads for 10 million instructions.

V. Results

We now present the results of using the SC for cache coherence verification. The SC can verify transactions once they appear on the shared bus and is unable to verify any transaction until it is seen on the bus. Hence, we present the fraction of cache coherence transactions that can be verified. Note that any unverified transactions will be verified in the near future when they will be seen on the bus, but after a certain number of elapsed cycles. Cache line sharing is a function of the benchmark used and thus, we analyzed the required size of the SC cache for each benchmark. Following this, results are presented when using a realistic SC cache size

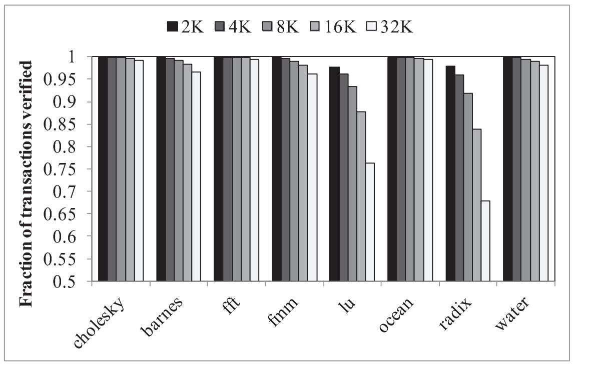

Fig. 4. Fraction of transactions verified for various cache sizes when using unlimited SC cache size.

to evaluate the effectiveness of the system. sizes, the above mentioned experiments are carried out for various cache sizes.

A. Unlimited SC cache size

We now present results showing the fraction of transactions verified for unlimited SC cache size and also the upper bound on the required SC cache size such that maximum verification is possible for all benchmarks. The results are plotted in Figures 4 and 5, respectively, for various general-purpose core cache sizes and various benchmarks. Figure 4 shows that in general, a very high coverage is obtained for smaller cache sizes and this reduces with increasing cache size, for the 10 million instructions that we simulated. This is intuitive since the smaller the cache, the larger is the number of cache conflicts and evictions. Also with smaller caches, a small proportion of the lines reside inside the L1 caches as compared to the total transactions seen on the bus. This also increases the fraction of verified transactions. It can be seen that other than *radix* and *lu*, all other workloads show greater than 0.9 coverage even when using a cache size of 32K. The reason for low coverage for *radix* is that it comprises almost 90% floating-point operations and involves very limited sharing of data. Most of the cache lines are exclusive and reside in the local cache for long periods of time. Cache miss rates were also observed to be small for both workloads, leading to fewer transactions on the bus, resulting in low coverage. It may be noted that the fraction of verified transactions asymptotically tends to 1 as the number of instructions executed increases. This is because the unverified transactions are always dependent on the size of the general-purpose caches for reasons just mentioned. But as the time increases, the number of transactions occuring on the bus is very high and the fraction of unverified transactions reduces to zero. Hence, even though some of the values in the plot suggest low verification ratio, it is due to the 10 million instructions that we ran. Increasing this number will increase the verification ratio. The number of cache lines required by the SC to log all entries is shown in Figure 5. In the worst case, no cores in the system will share lines and the cumulative sum of the lines occupied by all cores may need to be stored. From the figure, it can be seen that barring the workloads *barnes, lu, water*, the amount of storage required is the cumulative sum of all cache sizes and hence the capacity requirement is

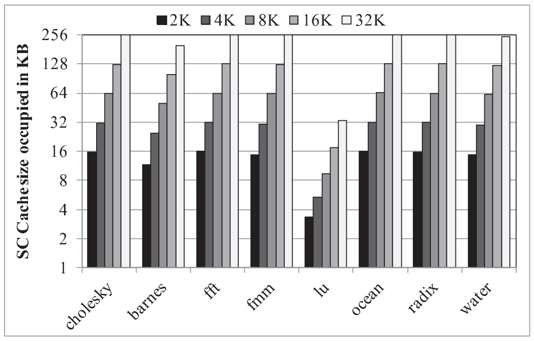

Fig. 5. Maximum size of the SC cache required such that all possible transactions seen on the bus are verified.

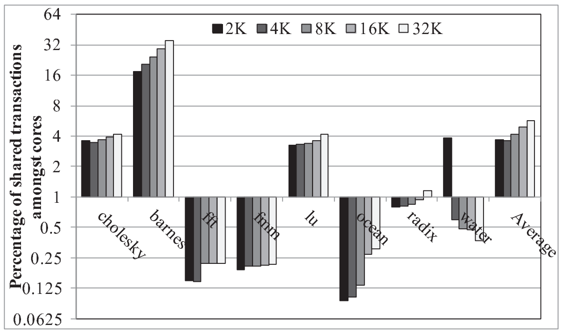

Fig. 6. Percentage of transactions that involve sharing cache lines amongst cores for each workload and various general-purpose core cache sizes.

very high (almost 256K for 32K cache sizes). This may imply that in order to achieve high verification rates for larger cache sizes, the SC cache size may needs to be prohibitively large. Fortunately, a key insight makes sure that this is not the case.

1) Discussion: From the results so far, we have seen that for verifying all possible transactions using the SC, the SC cache size may have to be equal to the sum of the cache sizes of all the cores in the system, for certain workloads. However, the transactions considered so far include lines that are exclusively held in the cache of a single core and those that are shared amongst the cores. We have seen in Section III-B, that this situation results in the worst storage efficiency for the proposed scheme. However, if something goes wrong in computing the cache line state while the line is held exclusively, it very rarely results in error. For example, faulty change of state of a line to M will result in write back when this line is evicted from the cache, but the data will not be corrupted. The unnecessary writeback will have a small performance penalty, but may be worth it if the tradeoff allows all other transactions to be verified. The more malicious case is when a fault causes a line in M state to move to a different state. Here, upon eviction the line will not be written to memory and the memory will no longer be up to date. This situation can be avoided by special encoding of the MESI states. For example, one hot coding for the four MESI states will ensure that no single bit error will ever go unnoticed. Thus, it may not be necessary to verify exclusively held cache line states. The more interesting but challenging case is the verification of line states globally across cores. We have observed that when the verification of the exclusive line states is excluded, the cache size requirement of the SC to log all transactions drops dramatically. Figure 6 shows the percentage of transactions that are shared amongst cores for various cache sizes. It can be seen that barring *barnes*, shared transactions for all other workloads account for a very small percentage of the total transactions. Hence, by dropping the verification of exclusive states, the size requirement for the SC cache can be reduced dramatically. In the next sub-section, we focus therefore, on verifying only the transactions that involve lines shared among cores.

B. Realistic SC cache size

We now present the results of our experiments to determine the capability of the SC to verify memory transactions in a more realistic scenario. Here the SC cache size is not unlimited and its associativity is identical to that of the caches of other cores in the multicore.

1) Percentage of transactions verified: We varied the size of the SC cache as well as that of the general-purpose cores from 2K to 32K. In total, we ran experiments for each workload for the 25 possible combinations of cache sizes of the SC and the general-purpose cores. We show the results for the workload which exhibited the worst case, i.e., *barnes* in Figure 7. For a small SC cache size, it is expected that the cache conflicts in the SC cache during transaction logging will be high and thus, the percentage of transactions verified will be smaller. For a fixed general-purpose core size, it can be seen that the percentage of unverified transactions drops drastically with increasing SC cache size, which is expected. The worst case of 59% transactions unverified is observed when the SC cache size is set to 2K and that of the general-purpose cores to 32K. However, this is not a realistic scenario as in general, it is expected that the SC cache size will be at least equal to that of the general-purpose cores. Looking at the data points in Figure 7 that represent equal SC and general-purpose core cache sizes, it can be seen that in almost all cases 100% transaction verification is possible. The only combination where this is not the case, is when the cache sizes are set to 32K for *barnes*, where 1.3% unverified transactions were observed. This is a very small fraction and this greatly increases our confidence in the capability of the proposed scheme. The other workload that showed missed transactions for same SC and general-purpose cache size is *ocean*, where 5.6% of the transactions were missed for a cache size of 8K. For the rest of the workloads, setting the SC cache size to 16K is enough to verify all transactions even when the general-purpose core caches are set to 32K. In Table II, the minimum size of the SC cache required for 100% verification of shared transactions amongst the 8 cores with L1 cache set to 32K, is shown. For a majority of the workloads, an SC cache of just 2K suffices. By excluding the verification of the the exclusive cache states, 100% of the transactions can be verified using realistic SC cache sizes. This result also shows that the proposal made by Borodin *et al.* [10] where a replica is maintained for each cache, is pessimistic, since there for 8 cores and 32K caches, the total checker cache size is 256K, as compared to 32K in the worst case for our scheme. Note that when using 32K cache

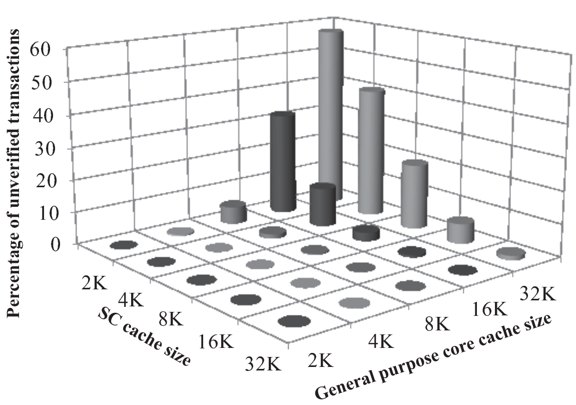

Fig. 7. Percentage of transactions unverified for the workload *barnes* for various combinations of SC and general-purpose core cache sizes.

TABLE II
MINIMUM SC CACHE SIZE REQUIRED FOR ≈100% TRANSACTION COVERAGE WHEN USING GENERAL-PURPOSE CORES WITH L1 CACHE 32K

Workload	Min SC L1	Workload	Min SC L1
cholesky	16K	lu	2K
barnes	32K	ocean	16K
fft	2K	radix	2K
fmm	2K	water	2K

size, only 1.3% transactions are missed in the worst case of *barnes*.

2) Time to verification time and performance penalty: The SC provides error detection. Error recovery is assumed to be in place using a checkpointing scheme [19]. If the error detection latency is larger than the checkpointing interval, the system state will be corrupted. Thus, the latency to error detection is important. In our experiments, we have observed that even in the worst case, the transaction verification latency is a few thousand cycles which is well within reasonable checkpointing intervals. The proposed scheme also results in increased bus traffic in the cases where cache lines in the general-purpose cores are overwritten without write back. We observed a 20% increase in bus traffic in the worst case, but this resulted in performance loss of less than 2%.

VI. CONCLUSIONS

In this paper, we have presented and evaluated a new centralized mechanism to verify the cache coherence transactions in a shared memory snooping bus multicore. The proposed scheme is based on the incorporation of a small and simple Sentry Core that can be assumed to be fault-free. The SC has access to the shared bus and it can log memory requests seen on the bus, in its cache. Since it is aware of the cache coherence protocol, based on the memory operation and the current state of the line requested, the SC knows the expected next state for the line. Whenever the same line is seen again on the bus, the SC compares the state of the line to what it computed and flags an error if a discrepancy is found. As the scheme depends on logging of transactions in a cache, its capabilities are determined by its cache size. Results were presented on the upper bound of the scheme for unlimited SC cache size. A realistic scenario was then presented where the SC cache was assumed to be similar to that of the general-purpose cores. Results were presented for various combinations of SC and general-purpose core cache sizes. These results indicate that in a realistic scenario of equal SC and general-purpose core cache sizes, >94% of the transactions can be verified. The performance penalty arising from the scheme was found to be less than 2% in the worst case. Our analysis also shows that using a centralized checker for cache coherence may result in far lower hardware overhead in terms of additional cache space required for checking.

REFERENCES

[1] Austin, T.M.; , "DIVA: a reliable substrate for deep submicron microarchitecture design," Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on , pp.196-207, 1999.
[2] Srinivasan, J.; et al.; , "The impact of technology scaling on lifetime reliability," Dependable Systems and Networks, 2004 International Conference on , pp. 177- 186, 28 June-1 July 2004.
[3] Borkar, S.; , "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," Micro, IEEE , vol.25, no.6, pp. 10- 16, Nov.-Dec. 2005.
[4] Renau, J.; et al., SESC Simulator, January 2005. http://sesc.sourceforge.net.
[5] DeOrio, A. et al.; , "Post-silicon verification for cache coherence," Computer Design, 2008. ICCD 2008. IEEE International Conference on , pp.348-355, 12-15 Oct. 2008.
[6] Ogawa, E.T. et al.; , "Leakage, breakdown, and TDDB characteristics of porous low-k silica-based interconnect dielectrics," Reliability Physics Symposium Proceedings, 2003. 41st Annual. 2003 IEEE International , pp. 166- 172, 30 March-4 April 2003.
[7] Benso, A. et al.; , "A watchdog processor to detect data and control flow errors," On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE , pp. 144- 148, 7-9 July 2003.
[8] Rodrigues, R. et al.; , "An Architecture to Enable Life Cycle Testing in CMPs," Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on , pp.341-348, 3-5 Oct. 2011.
[9] Saxena, N.R.; McCluskey, E.J.; , "Control-flow checking using watchdog assists and extended-precision checksums," Computers, IEEE Transactions on , vol.39, no.4, pp.554-559, Apr 1990.
[10] Borodin, D.; Juurlink, B.H.H.; , "A Low-Cost Cache Coherence Verification Method for Snooping Systems," Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on , pp.219-227, 3-5 Sept. 2008.
[11] Cantin, J. et al; "Dynamic Verification of Cache Coherence Protocols". ISCA Workshop on Memory Performance Issues, 2001.
[12] Fernandez-Pascual, R. et al.; , "A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on , pp.157-168, 10-14 Feb. 2007.
[13] Woo, S.C. et al.; , "The SPLASH-2 programs: characterization and methodological considerations," Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on , pp.24-36, 22-24 June 1995.
[14] Borkar, S. et al. Platform 2015: Intel Processorand Platform Evolution for the Next Decade. Technology@Intel Magazine, 2005.
[15] www.intel.com.
[16] www.AMD.com.
[17] Meixner, A.; Sorin, D.J.; , "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on , pp.145-156, 10-14 Feb. 2007.
[18] Meng Zhang; et al.; , "Fractal Coherence: Scalably Verifiable Cache Coherence," Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on , pp.471-482, 4-8 Dec. 2010.
[19] Sorin, D.J.; et al.; , "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on , pp.123-134, 2002.
[20] Hennessy, J.; and Patterson, D.; , "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 3 edition, 2003.