Parallel Assertion Processing using Memory Snapshots

Junaid Haroon Siddiqui, Muhammad Faisal Iqbal , Derek Chiou
The University of Texas at Austin
{iqbal, jsiddiqui, derek}@ece.utexas.edu

***Abstract*—Assertions are valuable in producing quality software but some assertions like those checking data structure invariants are costly by nature. The authors of this paper propose that costly assertions be queued up for later execution by a free processor. The state of data is preserved using a memory snapshot. Such a scheme will extract further parallelism from a given application and improve its speed up on a parallel machine.**

Three schemes for end user programs to use memory snapshots are discussed. One is based on memory mapping like *mmap*. Second scheme uses unused virtual address space bits for snapshot identifier. The third scheme uses a software library. Merits and demerits of user interface schemes are identified.

Three methods for backend implementation of memory snapshots are discussed. One is based on physical address space versioning. It proposes modifications to memory controller in a way similar to transactional memory. It goes on to introduce a snapshot aware cache coherence protocol. The second method is based on virtual address space versioning and suggests kernel modifications. The last method is totally supported in a user library. Advantages and disadvantages of different implementations are compared.

Evaluation is not extensive, given the large matrix of implementation possibilities. However basic memory controller snapshot capability is implemented in *M5* simulator. Additionally, a software library approach is implemented. Most valuable results of this evaluation are the pros and cons of various approaches discussed. The feasibility study of various implementation schemes is also an important groundwork for future research on memory snapshots. Based on these experiments, the authors believe that memory snapshot capability is a powerful tool that allows parallelism where it is not natural without such a capability.

***Index Terms*—Assertions, Parallelization, Memory Snapshots**

I. Introduction

Commodity machines are now getting dual and quad core processing power. Industry is targeting many more cores on every desktop. The 80 core prototype of Intel [1] is one such effort. This creates a big challenge for software writers who are accustomed to sequential programming. First efforts to face this challenge were introducing explicit thread creation and management libraries like pthreads [2]. This is the only widely adopted scheme but has many drawbacks. Several new parallel programming models are proposed to address these drawbacks. One example of such a language is cilk [3].

Whatever parallel programming model is used, efficiency usually decreases with an increase in speed up. According to Eager, "Along with an increase in speedup comes a decrease in efficiency: as more processors are devoted to the execution of a software system, the total amount of processor idle time can be expected to increase, due to factors such as contention, communication, and software structure" [4]. Due to this, any further parallelism that can be extracted would improve efficiency and speed up. The current research body has a lot of material on compile time extracting of parallelism from loops [5], [6], [7]. This paper is also concerned about extracting parallelism but is focused on the single domain of assertions. This allows domain-specific optimizations and a scheme for removing dependencies on the code following the assertion.

Use of assertions allows catching bugs early, resolving them quickly, and consequently improve the quality of software [8]. Increased density of assertions in the code results in decrease in bug density [9]. Experience demonstrates that designers can save up to 50% of debug time using assertions and thus improving the time to market [10]. Assertions are also useful in production software. Many modern software applications have a facility to report back failed assertions. The developers use this information to remove the bug, resulting in improved customer satisfaction and quality of future versions.

Some assertions are simple checks but some are costly validations of data structure invariants. Literature refers to these checks as *repOk* [11]. The performance penalty of such assertions limits their use in production software and sometimes in debugging sessions too. Traditionally this high overhead is reduced by limiting the use of assertions [12] and thus compromising the quality. The repOk method for a simple binary tree is shown in Figure 1. A binary tree should be acyclic and every node should have distinct children. Checking these properties is linear in time and space. These invariants should be evaluated whenever the data structure is altered. However, with these assertions present, binary tree performs worse than a simple array.

The authors propose that costly assertions like these be queued up for later execution by an idle processor. Since idle time increases with parallelism [4], we can potentially run these assertions at no added cost. The programmer's view of assertions is slightly altered. The program is stopped when an assertion fails, but at a point later than the assertion. The debugging benefits can still be achieved, because the exact assertion that failed is identified. However in production environments, some assertions may need to be evaluated sequentially if they precede some critical non-volatile change. To solve this problem, the authors suggest that this scheme be improved in the future to support rollback.

The biggest challenge in this scheme is that assertions

```
class BinaryTree {
  Node* root; // root node
  static class Node {
    Node* left; // left child
    Node* right; // right child
    // data
  }
  bool repOk() {
    std::set<Node*> nodes;
    return repOk_node(root, nodes);
  }
  bool repOk_node(Node* n, std::set<Node*>& nodes) {
    if(n->left == n->right) return false;
    if(n && nodes.find(n) != nodes.end())
      return false;
    nodes.insert(n);
    return repOk_node(n->left) && repOk_node(n->right);
  }
};
```

Fig. 1. repOk method for a Binary Tree

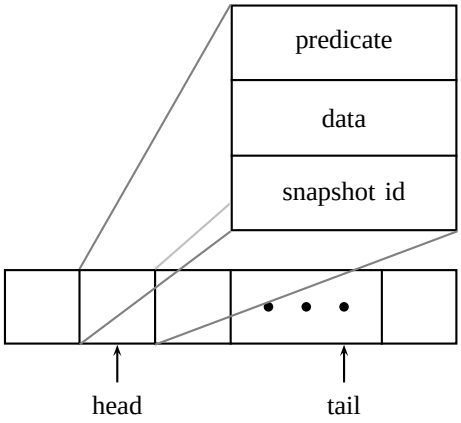

Fig. 2. Structure of Assertion Task Queue

require the state of memory as it was, when the assertion was queued. The authors propose an implementation of memory snapshots to make this possible. A low cost snapshot will be taken when the assertion is queued. This old data is used for assertion processing on the idle processor, after which the resources for this snapshot are freed. This makes it possible to have optional assertions which only run when sufficient idle cycles and snapshot resources are available. Or compulsory assertions that block the main thread, and wait for previous assertions to complete and make resources available.

The performance of this scheme depends on the overhead of taking, maintaining, and releasing a snapshot compared to the overhead of assertion itself. The snapshot overheads in turn depend on the characteristics of the main application itself. The schemes proposed below use copy-on-write, and if more copies have to be made, more resources will be occupied and performance will be worse. If the writes are infrequent, the overhead will be much lower. This means that applications have to be individually evaluated, where this scheme gives a performance boost.

The rest of the paper is organized as follows. Section II gives a step by step overview of the proposed scheme. Section III discusses three ways that memory snapshots can be used. Section IV details three implementations of memory snapshots. Preliminary evaluation of two of these schemes, some basic results, and feasibility study for more evaluations is contained in Section V. Section VI concludes along with possible future work in this direction.

II. Parallel Assertion Processing

The steps for parallel assertion processing are described in two parts; the steps taken when an assertion point is reached in main code, and the steps taken when an idle processor picks up the assertion task. The communication medium between the two entities is an assertion task queue. The first section lays out the queue structure, the next two sections list the steps involved, while the fourth section discusses schemes to allocate processors to do assertion tasks.

A. Assertion Task Queue

The assertion task queue is implemented as a fixed size circular buffer. This enables lock-free insertion when there is a single main thread producing assertions. When the original application is already parallel, there should be one queue per thread so that no locking is involved when queuing an assertion. The performance effect of having a single queue with locking, or multiple queues without producer locks are not investigated in this paper.

Structure of assertion task queue is shown in Figure 2. Each entry contains the address of a predicate routine, data to be passed to that routine, and a snapshot identifier. The number of entries in this queue bounds the number of outstanding assertions. However each entry needs a snapshot so there is no reason to have more queue locations than supported snapshots. The space overhead of this queue is therefore negligible.

B. Steps at Assertion Point

There are two steps to be taken at an assertion point. A *take_snapshot* step and a *queue* step.

1) Take Snapshot: A memory snapshot is taken. There is an identifier associated with each snapshot. Since memory snapshots take resources, it is possible that these resources are not available right away. The caller has the option of waiting for resources (blocking snapshot), execute the assertion sequentially, or ignoring the assertion altogether. The first two are schemes to make *compulsory assertions*, while the last option results in *optional assertions*.

2) Enqueue Assertion Task: A predicate function, a pointer to some data structure, and a snapshot identifier are queued in assertion task queue. Whatever the number of assertion task processors, there is no locking needed to insert in the queue as there is a single producer.

The main thread will proceed execution after queuing this assertion and may modify the data structure being asserted and may issue more assertions.

C. Steps at Assertion Task Processing

When a processor decides to take up an assertion task, it does a *dequeue* step, an *assert* step, a possible *halt* step, and

finally a *release_snapshot* step.

1) Dequeue Assertion Task: An entry is dequeued from assertion task queue containing a predicate function, a pointer to be passed to this function, and a snapshot identifier. If the system has multiple assertion task queues, then they all have to be queried for an available task. If they all have tasks there is a fairness issue and a performance issue in determining the order to process them. The performance issue exists because resource requirements of snapshots grow as time passes.

2) Evaluate Assertion: The predicate function is then called with the given pointer and snapshot identifier. The predicate function has to use one of the schemes discussed in section on user interface to access data in the old snapshot. There is a performance penalty to doing this, which means assertions are now running slower than they were originally running. However the contention is that they are using idle time, and the main thread has less work to do now, and therefore the overall performance will improve.

3) Possible Halt: Execution has to be halted if the assertion fails. There is an issue of semantics here. Programmers are accustomed to the semantics of assertion halting right at the assertion point. For example, they may see more console output from after the assertion point. There are two issues to this. Most important is finding the exact point for debugging, and the given mechanism still points the exact assertion point that failed even though it was evaluated much later. There is another issue of undesirable side effects from after the assertion point. The authors believe that given this snapshot capability it is easy to implement a memory rollback facility. Such a facility would eliminate side effects in memory. If there are possible side effects on external media or memory, such an assertion should be executed sequentially. There is an added advantage to rollback facility. We can modify the current halt step which a more sophisticated rollback_and_halt step that would make an attached debugger break at assertion point, rather than assertion processing point. Rollback is not discussed any further in this paper and is left as a future work.

4) Release Snapshot: The last step is to release the snapshot. This will free any resources occupied by the snapshot due to modifications from the main thread.

Depending on the assertion task processor allocation model, the thread can go on and serve more assertions, or go back to some other real work that is now available for it to perform.

D. Processor Allocation for Assertion Tasks

Based on the parallelism model adopted by the application, an appropriate model can be chosen for processing assertion tasks. One model is creating one or more dedicated assertion processing tasks. These tasks should preferably be bound to processors so that caching benefits can be used. For example, if a data structure invariant is repeatedly checked, most of the data can be found in processor cache. This binding of tasks to processors is called processor affinity. Popular operating system provide calls for this purpose [13], [14].

Another model is that of a work queue based system. In such a system, assertion can go in the same work queue possibly with low priority. Another model to use in custom event driven systems, is that a thread that picks and serves an event serve its own assertions after signaling that the event is served. The thread can have cache benefits even on the first run. At the same time, any dependent events can start in parallel since completion has been signaled.

There can be many other models. The critical aspect is separating assertion point from assertion processing and as soon as that is done, models for parallel assertion processing best suited for a given application are not hard to come up with.

III. User Interface to Memory Snapshots

The scheme proposed above needs a *take_snapshot* operation, a *release_snapshot* operation, and a mechanism to access data within a snapshot. This section discusses multiple semantics for these operations, and compares them.

A. Accessing Snapshots by Memory Mapping

A flexible scheme for taking and accessing snapshot is a *mmap* like system call with the following signature.

```
void* mmap_snapshot(void* p, size_t len);
```

This would form a snapshot of `len` bytes starting at `p`, and make it accessible at the returned pointer. This scheme gives the flexibility of making snapshots of a specific area. However, dynamically allocated node based structures are not contiguous. The best in that case would be to snapshot the whole heap. No separated snapshot identifier is needed. The returned pointer can serve this purpose.

This scheme however requires that the kernel setup page tables for this new area and set them for copy-on-write behavior. This means an overhead at the time of taking a snapshot. Additionally there is overhead when the main thread changes its data causing copy-on-write. The assertion overhead has to be much more to offset these overheads.

B. Accessing Snapshots using Virtual Address bits

Another scheme for taking snapshots of whole memory and avoiding overhead of setting up new page tables is to use virtual address bits. This means that the virtual address space is divided in parts, where one part is the current version, and the rest of the parts are snapshots of the current version.

In 32-bit processors, the virtual address space is already scarce, but in 64-bit processors, there is much more space available. However no 64-bit processor has a 64-bit virtual address space. For example, Alpha EV5 [15] has a 43-bit virtual address space. If rest of the 21 bits in 64-bit pointers could have been used, they are enough for a snapshot identifier. Unfortunately, the Alpha ISA also uses this information and encodes only 43 bits in addresses contained in instructions. The ISA needs to be changed to support an alternate instruction encoding.

The other approach is to use some bits in the valid address space. This is the only scheme applicable in 32-bit machines. This means that the operating system has to change its process address space model. This is a significant change in the kernel. Many user applications also make assumptions about

```
class snapshot_manager {
  snapshot_id take(bool block);
  void release(snapshot_id id);
  void modify_notification(void* p, size_t len);
  void* read_ptr(void* p, snapshot_id id);
};
class assertion_queue {
  assertion_queue(snapshot_manager& sm);
  bool queue(predicate_type pred, void* data, bool block);
  bool process(bool block);
};
```

Fig. 3. Software Library for Parallel Assertion Processing

the address space. However, ignoring those applications, it is possible to support this in the kernel.

When a page fault occurs in the snapshot area of the virtual address space, the kernel has to perform a mapping. However the kernel can calculate this mapping at fault time. This means that the cost of setting up page tables is transferred from main thread to assertion processing thread. If this scheme is compiled with physical memory snapshot implementation below, then no fault will be generated and the kernel will not be bothered.

C. Accessing Snapshots from a Software Library

There is a pure software solution to this problem. The overheads however are much higher, and there is an added cost on the software developer. It is useful therefore in fewer circumstances. However being in software, it is available everywhere.

Two classes, `snapshot_manager` and `assertion_queue` are proposed. There methods are listed in Figure 3. Class `assertion_queue` uses `snapshot_manager` to implement parallel assertions. Other applications that want to use snapshots can use `snapshot_manager` directly.

To implement copy-on-write in software, the data structure has to inform the snapshot manager of any changes its going to make. For example a binary tree would inform it of any nodes it is going to delete, or any nodes whose internal pointers it is going to change. This needs changes in the data structure but is not hugely demanding. There are fairly limited and easily identifiable places where data structure is altered.

Snapshot data is accessed using `read_ptr` which may return an old copy or the latest data if it is still unchanged. There is a synchronization issue here that the assertion thread does not want the main thread to modify data *while* it is reading it. Calling `read_ptr` poses an overhead but this overhead is only for the assertion thread. Overheads for the main thread are `modify_notification` and the synchronization issue when the assertion thread is accessing latest data. More about these issues are discussed in Section IV-C.

IV. Implementation of Memory Snapshots

Three schemes are discussed to implement memory snapshots. One in which the hardware supports snapshots using physical address space bits, one in which the kernel supports snapshots, and the third is a pure software solution.

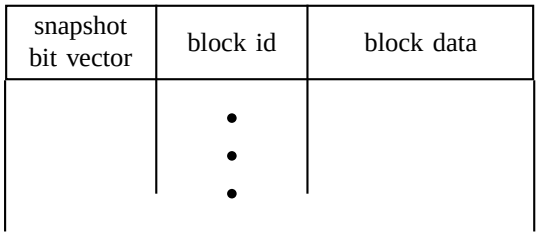

Fig. 4. Structure of Snapshot Cache

A. Snapshots of Physical Address Space

This scheme uses bits in physical address space for snapshots. This is again restrictive for 32-bit machines, as maximum allowed memory will decrease. However for 64-bit machines, there are enough bits that few can be used for snapshot identifier.

A mechanism for taking and releasing snapshots needs to be introduced. Being in hardware, the maximum number of snapshots has to be fixed. The following details are described for 16 snapshots. Four high order bits will be reserved for snapshot identification. A 16-bit register in memory controller remembers which snapshots are taken and which are released. For example if bit X is set, then snapshot X has been taken. This register is referred below as the *snapshot register* The snapshot register will be accessed using memory mapped I/O to take and release snapshots.

For copy-on-write behavior, the memory controller has to divide memory in blocks, referred here as just *blocks*. Experimentation can determine the granularity best suited to most applications. Potentially it can be as small as a cache line and as large as a page. However larger blocks result in longer copy-on-write operations while smaller blocks result in more copy-on-write operations.

Other than snapshot register, the memory controller keeps a block cache for old copies of data, called *snapshot cache*. This is similar to the way transactional memory implementations keep a second copy [16]. Unlike Herlihy's transactional cache contained in the processor, this scheme keeps the transactional cache with the memory controller. Snapshot cache structure is shown in Figure 4. The purpose of the fields is as under:

- The snapshot bit vector tells which snapshots contain the state of memory given in this entry.
- Block of memory this entry is about is the *block identifier*.
- Old copy of data is contained in *data* field.

Behavior of memory controller is changed as follows.

- When a snapshot is taken, the memory controller sets the bit in snapshot register.
- When memory is written, the memory controller finds the affected block and finds the snapshots that already have this block in snapshot cache. For snapshots that do not have this in snapshot cache, a new entry is made. It contains these snapshots in the snapshot bit vector, the memory block identifier, and the actual current data of memory.
- When a snapshot is released, the memory controller has to traverse the snapshot cache, reset the corresponding bit in every entry, and then reset the corresponding bit in

snapshot bit vector	tag	state	data
011101000	X	Shared	· · ·
000000001	X	Exclusive	· · ·

Fig. 5. Cache entry duplicated on being written to

snapshot register. This is a long operation but it can be done lazily. However if someone tries to take a snapshot with the same identifier, it has to be blocked until the lazy release operation finishes.

So far this scheme needs no help from caches. Two copies of same memory from different snapshots have different tags in the cache. The advantage of this scheme is that any cache coherence protocol and unmodified processor core can be used with this memory controller. The disadvantage, on the other hand, is that the potential advantage of hitting snapshot's data in the cache is now gone. It can be argued that snapshot is being taken for another processor to use, so there is no advantage to hitting the snapshot data in the cache. This is however not true, because the other process may have *touched* the data some time ago just for performance. Also in the case of parallel assertion processing, an assertion thread may have the data cached from the last time it processed the same assertion. Therefore it is always useful to have a snapshot aware cache.

To solve this problem, support for snapshots is added to caches. The cache is aware of the snapshot identifier bits in the physical address. It contains its own snapshot register as well. Any access to the snapshot register will still go on the bus for other caches and memory controller to update their state. On a non-bus system, broadcast has to be used. Each entry in the cache is modified to contain a snapshot bit vector implying the snapshots for which this entry is valid. This is pictured in Figure 5.

The behavior of cache is modified as follows:

- When a snapshot is taken, the corresponding bit is marked in every entry which is current. A current entry means that it is in snapshot 0, which is the current state of memory.
- When data is read or written, the snapshot bits are separated, the tag is separated which does not contain the snapshot bits. Then an associative lookup is done using tag for entries that have the corresponding bit set.
- When data is written and there is any bit other than the current bit set in the cache line entry, a duplicate has to be made in the cache. This can cause a write-back. This duplicate will only have the current bit set, while the old copy has it cleared as shown in Figure 5.
- When a snapshot is released, the corresponding bit is cleared in every entry. This can again be a lazy operation, as discussed for memory controller above. Any entry with all bits cleared is now a free entry.

Giving preference to new copies in cache replacement algorithms can potentially improve performance. This is because memory controller's task is simplified and its limited snapshot cache is better utilized.

The advantage of physical memory versioning is that most of the operations are done in hardware and can be optimized to do their specific tasks. There are several disadvantages as well. The snapshot cache is limited in size and at some point the memory controller has to discard some snapshots or block further writer. The later is not practical as it will stall the processor altogether. Another approach can be to run a special handler once it reaches near its capacity.

Another problem with snapshots of physical memory is that performance of one application is dependent on another. Even though one application cannot change or access data in another application, its changes are causing new entries in snapshots made by the other. The logical conclusion from this is applications are truly interested in snapshots of their own virtual address space and not the global physical address space.

A related problem with snapshots of physical address space is caused by demand paging. If something is on the disk, and we are taking snapshots of physical address space, it does not become part of the snapshot. There can be some ways around it with kernel support. However kernel support directs us to a more direct solution of making snapshots of virtual address space.

B. Snapshots of Virtual Address Space

Operating system takes the central role in supporting snapshots of virtual address space. It can support them with either of the three schemes discussed in Section III. Hardware support can help in the performance of operating system.

To support a memory mapping based scheme, the operating system would find and map a new range of memory to the same physical range. It would then set the range for copy-on-write behavior.

Supporting a scheme of virtual address space bits is similar. However it needs to restrict its use of virtual address space to the portion where snapshot identifier is zero. Other than restricting its use of virtual address space, it has to setup the address translation for snapshots. It can do by either setting up all page tables right away like in a memory mapping scheme, or like in *fork*. Alternately, page table can be setup as needed on page faults. This is possible since the virtual to physical mapping for snapshots is identical to current memory until something is modified.

The problem however is that setting the complete address space for copy-on-write is a significant overhead, both in setting it up, and for future write operations by the main thread. This is a special concern for operating systems using copy-on-write for *fork* operation.

This gives rise to the third option of kernel support for a software library. Two things that kernel has and user software does not have are address translation control, and copy-on-write. If software library has access to these, it can support a mechanism where only the necessary pages are set for copy-on-write, and the standard translation mechanism is used to access old copies.

C. Snapshots User Library

This section discusses a software implementation of interfaces given in Section III-C. The `assertion_queue` can be implemented using a simple queue with consumer locks. If there are more than one producer, producer locks can be used or alternately one queue per producer can be used.

The `snapshot_manager` keeps the snapshot cache in a software map. The unit of storage in snapshot cache is not fixed. Rather it is the same as passed to `modify_notification`. Thus if different sized nodes are reported to `modify_notification`, they will be copied using dynamic allocation and stored in the map. The actions of snapshot manager are as follows:

- When a new snapshot is taken, it just remembers it in a bit vector.
- When a modify notification is given, it sees if the given data has already been cached by active snapshots. It makes a new cache entry for snapshots that do not already have this data cached.
- When a read request is received, it sees if it has data for the requested version. It returns the cached data if available, or else returns the same pointer. This means that the data has not since changed. There is a synchronization issue here that the data may be changed after returned from this function. One way to handle this is blocking the main thread until a new `read_done` function is called. Another way is to have `read` return actual data and not pointers. This results in more calls to `read` but can possibly keep the main thread a bit faster.
- When a snapshot is released, every entry in cache should have its bit cleared. Any entries stored only for this snapshot are freed. Lastly, its bit is reset in the snapshot bit vector.

Major advantage of this scheme is that no changes in operating system or hardware are required. However, there is a big performance overhead due to lookups of a software cache. A hybrid software-kernel scheme should reduce these overheads significantly, as discussed in the last section. Another potential drawback is that missing a modify notification can result in corrupted snapshots. In parallel assertion processing, it can mean finding bugs harder, rather than easier.

V. Evaluation and Results

Two schemes are evaluated. A pure software approach and a memory controller based scheme are investigated. In the software approach the classes discussed in Section III-C are implemented. Old snapshots can be correctly accessed. Bounded Exhaustive Testing [17] of a Binary Search Tree was then conducted with assertions on every insertion. This kind of application is not suited to snapshots as the changes are very frequent putting a strain on snapshot cache. Also the only real work the application is doing is asserting. Due to the nature of application, there was a significant slowdown instead of speedup when assertions were executed in parallel. The main application was blocking for assertions to finish, as it was asserting at a high rate. Assertion processing is slower due to accessing the snapshots. And due to this dependency the main application is much slower.

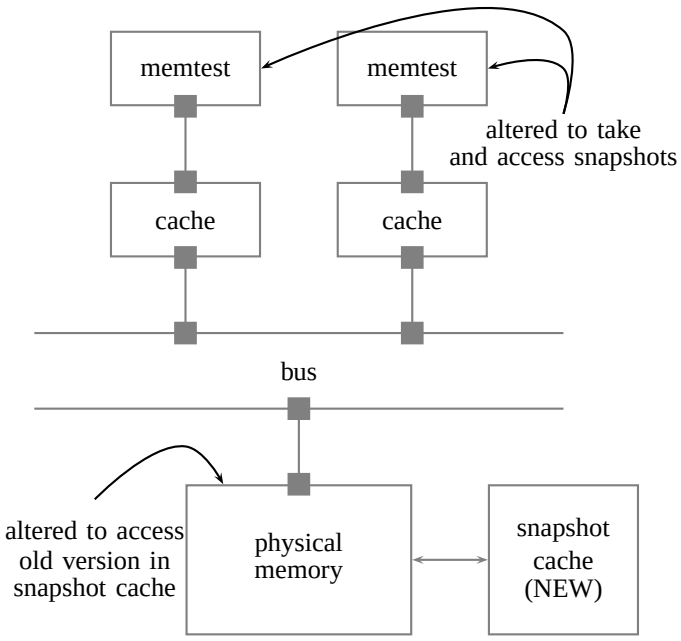

Fig. 6. Memory Snapshot Capability in M5

The memory controller based scheme is implemented in *M5* [18] . Two other simulators were thoroughly investigated for implementation but they did not allow sufficient infrastructure to implement this new controller. Gems [19] provides a memory timing model, but does not provide a functional model where a different data can be provided by changing code. That part was contained in proprietary Simics code. The other option investigated was *PIN* instrumentation tool [20] which works by using the x86 trace flag. However x86 instructions can generate more than one memory access per instruction, and the data provided by each of these accesses has to be controlled. Finally M5 gave this flexibility. However when an ISA is used on top of M5, and a normal program is run, virtual addresses have to be used. Using virtual addresses poses new issues discussed above, which require the kernel to be modified. To work around it, a special module was used that replaced the ISA and worked directly with the memory hierarchy. Using this module and a new physical memory module with an attached snapshot cache, the authors were able to experiment taking, releasing, and accessing snapshots on M5. Figure 6 shows modules from M5 that are involved in the evaluation. Gray boxes are ports which connect different components together. Changes done for implementing memory snapshots are highlighted.

There are lots of different components that need to be in place for an extensive performance evaluation. Even then, the behavior of application including its current speedup and efficiency determine the results to a significant degree. This paper is a start in identifying the issues and possible solutions in this area. The authors believe that further research in this direction will give more quantitative results.

VI. Future Work and Conclusions

Programmers avoid costly assertions like validating data structure invariants. However these assertions are as useful in catching bugs early as the simpler checks that programmers do use. Extracting costly assertions to be executed in parallel can improve the parallel speed up of many applications. This is important as current generation software is not scaling to next generation parallel hardware. The exact performance benefit depends on many factors including cost of queuing and maintaining a snapshot, cost of the actual assertion, idle parallel processors. The proposed scheme is most beneficial when the cost of assertion is high and idle processors are available, whereas cost of queuing and maintaining a snapshot is low.

Our evaluation work shows the feasibility of implementing a snapshot scheme, both in software and hardware. However they are useful in different scenarios. We hope that a future work would give a workload characterization where different schemes are most useful.

There are numerous potential uses of a generic snapshot scheme. The current paper is concentrated around parallelizing assertions. We envision some interesting application if support for rolling back to snapshots is added, which should not be very hard. One of these applications is *reverse debugging* where one can step back in the debugger. In fact a more interesting and less resource hungry scheme would allow rollback per stack frame. Other potential uses are memory transactions and software controlled state rollback. The authors will be investigating these potential uses in the future.

Another concept related to snapshots is having multiple active versions. Snapshots are passive versions of memory as no *writes* are allowed. Such a scheme can allow a new parallelization model where threads work independently on the same memory, and their work is merged later. This model is similar to software development model where code is developed independently and merged later.

References

[1] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, A. Singh, T. Jacob, A10, S. Jain, A11, S. Venkataraman, A12, Y. Hoskote, A13, N. Borkar, and A14, "An 80-tile 1.28tflops network-on-chip in 65nm cmos," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, 2007, pp. 98–589. [Online]. Available: http://dx.doi.org/10.1109/ISSCC.2007.373606

[2] "pthreads: The open group base specifications issue 6, ieee std 1003.1, 2004 edition." [Online]. Available: http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996. [Online]. Available: http://citeseer.ist.psu.edu/blumofe95cilk.html

[4] D. L. Eager, J. Zahorjan, and E. D. Lozowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. 38, no. 3, pp. 408–423, 1989.

[5] Y. Lin and D. A. Padua, "On the automatic parallelization of sparse and irregular fortran programs," in *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. London, UK: Springer-Verlag, 1998, pp. 41–56.

[6] M. J. Martín, D. E. Singh, n. Juan Touri and F. F. Rivera, "Exploiting locality in the run-time parallelization of irregular loops," in *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 27.

[7] M. Arenaz, n. Juan Touri and R. Doallo, "A gsa-based compiler infrastructure to extract parallelism from complex loops," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2003, pp. 193–204.

[8] S. McConnell, *Code complete: a practical handbook of software construction*. Redmond, WA, USA: Microsoft Press, 1993.

[9] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical study," *IEEE ISSRE'06*, 2006.

[10] Y. Abarbanel, I. Beer, L. Glohovsky, S. Keidar, and Y. Wolfsthal, *Computer Aided Verification*. Springer Berlin/ Heidelberg, 2000.

[11] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: A tool for generating structurally complex test inputs," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 771–774.

[12] T. L. Dahlgren and P. T. Devanbu, "Improving scientific software component quality through assertions," *ACM SE-HPCS*, May 15 2005.

[13] *MSDN Library: MSDNSetThreadAffinityMask Function*. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms686247(VS.85).aspx

[14] *Linux User's Manual: taskset*. [Online]. Available: http://www.linuxcommand.org/man_pages/taskset1.html

[15] "Alpha 21164 microprocessor: Hardware reference manual," Digital Equipment Corporation, July 1996. [Online]. Available: http://h18002.www1.hp.com/alphaserver/technology/literature/21164hrm.pdf

[16] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pp. 289–300, 16-19 May 1993.

[17] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 133–142.

[18] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.