On the Definition of Sequential Consistency

Ali Sezgin* Ganesh Gopalakrishnan

Abstract

The definition of sequential consistency is compared with an intuitive notion of correctness. A relation between what actually happens and what seems to happen is argued to be necessary. An alternative definition is proposed through the use of an infinite family of sequentially consistent transducers.

Key words:
Shared memories, Sequential consistency, Formalization

PACS:

1 Introduction

The behavior of a shared memory system is essentially a relation between a program and the set of executions it might generate. These relations are known as shared memory models. The larger the number of possible executions per program a shared memory model allows, the more optimizations in its implementations it is likely to permit. As such, the formal definition of a shared

* Corresponding author

Email addresses: sezgin@cs.utah.edu (Ali Sezgin), ganesh@cs.utah.edu (Ganesh Gopalakrishnan).

Preprint submitted to Elsevier Science 17 October 2004
memory model as a means to remove any ambiguity from its description is crucial.

Many shared memory models have been proposed in the past. One of the first to be formulated, and which has enjoyed much interest from programmers, designers and researchers in formal verification, sequential consistency [1] is defined as:

[A shared memory system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

So, sequential consistency requires that the memory behaves as if it responds to a single user whose instruction stream is an interleaving of the instructions of all the users such that this interleaving respects the order of instructions per user. What happens in real time and what seems to have happened logically are usually represented as two strings over responses (completed instructions). The former is called the *temporal execution*; the latter is *logical explanation*.

The original definition, given in an informal manner, seems to have been fairly well-understood, although certain discrepancies, probably contextually negligible, can be seen due to usually implicit assumptions about the system under consideration.

Despite the apparent consensus, we believe that the notion of correctness, the main motivation for the introduction of sequential consistency, is not adequately captured by its definition. We argue that the complete detachment between a temporal execution and its logical explanation poses more problems
than previously anticipated. In this paper, we will discuss this point strictly from a mathematical perspective; that is, we will not consider ourselves restricted to what has come to existence or what is likely to exist in the future as far as a shared memory system is concerned. Instead, our sole concern is whether the definition of sequential consistency permits intuitively wrong systems to be deemed sequentially consistent.

2 The Game of Verification

Let us first call the formal verification of sequential consistency for a given shared memory system as the verification of sequential consistency. In our framework, a shared memory system is parameterized over the set of processors, or users, the system interacts with, the set of addresses the system possesses, and the set of data values each address can contain, the sets $P, A$ and $D$, respectively.

A memory system either receives an instruction from a processor, performs internal operations or generates a response for a processor. For simplicity, we will assume that there are two types of instructions (responses): read instructions (responses) and write instructions (responses). A read instruction issued by processor $p$ to address $a$ is represented by $?a_p$. A write instruction issued by processor $p$ to address $a$ of data value $d$ is represented by $!a_p^d$. A read response is represented by $a_p^d?$ where $p$ and $a$ are as above, and $d$ is the data value returned for the read. A write response is represented as $a_p^d!$ where $p$, $a$ and $d$ are as above.

In our hypothetical setup, there is a team of experts on verification of sequen-
tial consistency. They are assumed to have a set of methods to decide whether a given finite state system is sequentially consistent or not\(^1\). However, they are not allowed to probe what goes within the shared memory system. The only observable information is the instructions issued to the system and the responses the system generates. They are also assumed to know the sets \(P\), \(A\) and \(D\) for the system at hand.

The task of this team of experts is to check whether a given memory system is sequentially consistent only making use of the information they are provided with.

3 The Deceiver

Let us define the following machine, called the *deceiver*. It has four main parts: the synchronizer, the decoupled unit, the serial unit and the filter.

The synchronizer checks the first \(|P| \cdot |A|\) instructions. If this many instructions are issued and there is exactly one write instruction for each address by each processor, the synchronizer gets into *villain* mode. Otherwise, it goes into *angelic* mode.

The filter selects which response is to be output once the synchronizer sets a mode of operation. If it is in villain mode, then the responses generated by the decoupled unit are output. Otherwise, the responses generated by the serial

\(^1\) It is worth noting that we do not share the belief that due to an undecidability proof in [2], the verification of sequential consistency is an undecidable problem for finite state systems; our argument on this will be published elsewhere. The reader is also referred to [3].
unit are output. It also has a storage of size $2|P| \cdot |A|$ to buffer the responses generated by the decoupled and the serial units until the mode of operation is selected by the synchronizer.

The decoupled unit consists of $|P|$ copies of a storage unit of $|A|$ addresses with each address capable of holding any data value from $D$. Each storage unit is uniquely identified with an element from $P$. When a read instruction $?a_p$ is input by the system, the contents of address $a$ in the storage $p$ is looked up and the response $a^d_p ?$ is generated where $d$ is the value returned by the storage unit. When a write instruction $!a^d_p$ is input by the system, the contents of the address $a$ in storage $p$ is changed to hold the new value $d$ and the response $a^d_p !$ is generated.

Finally, the serial unit consists of a single storage of size $|A|$, where again each address can return any value from $D$. This storage unit behaves like the ones in the decoupled unit with the exception that the processor index is abstracted; all read and write operations are performed on this single storage unit.

4 Deceiver is Sequentially Consistent

First of all, it is easy to see that the deceiver is not a correct memory system. When it goes into the villain mode, it behaves as a non-coherent memory system where no processor is able to see the updates done by any other processor in the system. Each processor reads only what it itself writes and nothing else. Therefore, in villain mode there is no communication among processors through shared memory which means that parallel algorithms developed for sequentially consistent systems will fail: a mutual exclusion algorithm will ei-
ther reach a deadlock or will result in the simultaneous access of the critical section by several processors; semaphores in general will not work. However, as we shall argue below, the team of experts are going to tag deceiver as sequentially consistent since behaviorally there is nothing wrong with it!

Now, if the machine is in the angelic mode, what is observed is basically a serial machine’s behavior. Each instruction is processed in the order it is input and the logical explanation, that is how the system seems to have behaved, is the same as the temporal execution, how the responses are temporally ordered.

If the machine goes into the villain mode after reading the first $|P| \cdot |A|$ instructions, each instruction again is processed in order. However, this time, the logical explanation is much different from the temporal execution. The logical explanation, in fact, is the juxtaposition of all the responses of each processor where no interleaving among responses of different processors is allowed. For instance, one logical explanation will first have all the responses for processor 1, then all the responses for processor 2, then for processor 3, and following this pattern up to processor $|P|$.

We will demonstrate the operation of a sample deceiver. We have $P = \{1, 2, 3\}$, $A = \{a\}$ and $D = \{0, 1, 2\}$.

Let us first consider the instruction sequence

$$!a_1^2, ?a_3, !a_2^1, ?a_1$$

Then, since the synchronizer will go into angelic mode after reading the second instruction, which is a read, the output of the deceiver will be

$$a_1^2!, a_3^2?, a_2^1!, a_1^1?$$
Hence, as can be seen, it is indeed behaving like a serial memory. It is a well-known fact that the serial memory is a sequentially consistent system; in fact, it was proposed as an operational model for sequential consistency.

Let us now consider the instruction sequence

\[ a_1^1, a_2^2, a_3^0, ?a_1, ?a_2, ?a_3, a_1^2, a_1^1, ?a_2, ?a_3 \]

This time, the machine will not output any response until the third instruction after which the synchronizer will go into the villain mode. So the execution will be

\[ a_1^1, a_2^2, a_2^0, a_1^1, a_1^1, a_2^2, a_3^2, a_1^2, a_1^2, a_2^1, a_3^2, a_2^2, a_2^2, a_3^2 \]

One possible logical explanation for this execution can be obtained by rearranging the responses as follows:

\[ a_1^1, a_1^1, a_2^2, a_2^2, a_1^1, a_1^1, a_2^2, a_2^2, a_1^2, a_1^2, a_2^1, a_3^2, a_3^2, a_3^2 \]

As was mentioned above, the first four responses are all the responses for processor 1, the next four responses are all the responses for processor 2, and the remaining responses all belong to processor 3.

It is not difficult to prove the following.

**Claim 1** All the executions that the deceiver generates belong to a sequentially consistent memory system.

Hence, our team of experts will conclude that the deceiver is sequentially consistent due to the absence of a program whose execution violates sequential consistency.
5 Proposal for a New Definition

First of all, we should admit that the deceiver is not likely to be developed by any designer who wants to keep his/her job. It is a highly unreasonable machine which we just used in order to demonstrate the mathematical inaccuracy in the definition of sequential consistency. However, as systems get more and more complex where no one designer will be capable to “know it all”, to be aware of every minute detail about the internal operation of the system, modes of operation which behaves like the decoupled unit can be possible. We cannot always depend on the sanity of the designer(s), especially when “we” represents a team of formal verification researchers. As such, the designer is always a collaborative competing agent; he/she should help in formalizing the design but his/her assertions about anything should be subject to scrutiny. In short, everything needs to be proved.

If we go back to the definition of sequential consistency, we can see that what it intends to carve out as sequentially consistent and what it mathematically defines are two different sets, as our “deceiver” machine exhibits. We believe that the major reason for this inequality is the absence of a constraining relation between the temporal order of execution and its logical explanation. How the responses are generated temporally and what the memory system seems to be doing logically should not be completely arbitrary.

Based on this observation, we believe that one has to be more specific about how much the temporal order and the logical explanation can be apart. Let us call a memory system logically bounded by \( f(i) \) if for any of its temporal executions, it has a logical explanation such that if a response occurs at the
\(i^{th}\) position in the execution, then that response must occur in a prefix of the logical explanation of size at most \(f(i)\). For instance, for serial memory we have \(f(i) = i\). This is due to the fact that the temporal order of execution and its logical explanation are the same for serial memories; that is, for the logical explanation equal to the temporal execution, the \(i^{th}\) response in the temporal order is also the \(i^{th}\) response in the logical order.

For another example, consider linearizability\([4]\). A linearizable system is sequentially consistent and satisfies an additional property: if a response is generated before another instruction, then the latter’s response cannot precede the former in the logical explanation. For linearizable systems, it is also assumed by definition that a processor can have at most one pending instruction\(^2\) per processor. We can then prove the following lemma.

**Lemma 2** For a linearizable system, we have \(f(i) = i + p - 1\) where \(p\) is the number of the processors.

**PROOF.** Let \(r_1\) and \(r_2\) be two responses generated during a computation such that \(r_1\) precedes \(r_2\) in temporal order. By the definition of linearizability, \(r_1\) can be placed after \(r_2\) in a logical explanation only if the instruction corresponding to \(r_2\) precedes \(r_1\) in temporal order. If this latter condition holds, we say that \(r_1\) can bypass \(r_2\). Since there is at most one pending instruction per processor at any point during the computation of the system, a response can bypass at most \(p - 1\) responses. This in turn means that the \(i^{th}\) response has to occur within the prefix of the logical order of size \(i + p - 1\).

\(^2\)An instruction which has been issued but whose response has not been generated.
To further illustrate this approach, note that for the deceiver, $f(i)$ is not bounded. As an example, consider the following sample execution of a system with one address and two processors.

\[a_1^1! \ a_2^2! \ a_1^1? \ a_2^2? \ a_1^1? \ a_2^2? \ldots\]

In this execution, the deceiver is in the villain mode and hence the decoupled unit is active. The logical explanation has to order first all the instructions of one of the processors. Without loss of generality, assume that it is the first processor. Then, the second response, $a_2^2!$ will not occur in the prefix of the logical order of size $s_1$ where $s_1$ is the number of responses belonging to processor 1. Since we can arbitrarily increase the number of instructions issued by processor 1, for every $n$, there will be an execution whose logical explanation will not have the second response in a prefix of size $n$.

However, neither the serial memory nor the notion of linearizability can be taken as alternative definitions for sequential consistency. Serial memory is too restrictive as it does not allow nondeterminism among different processors. Neither of the two allows a processor to have more than one pending instruction. These are properties which are almost always violated by current systems as satisfying them will be detrimental to the performance of the shared memory. Making use of the work presented in [3], we employ $SC$ machines to propose an alternative definition for sequential consistency. In what follows, we will assume that the notation of [3] is known; the reader is kindly referred to the chapters 2 and 3 of the mentioned work for relevant definitions.

**Lemma 3** For an $SC(j, k)$ machine, we have $f(i) = i + k \cdot p - 1$ where $p$ is the number of processors.
PROOF. First, let us note that the logical order corresponding to a computation of an SC machine is given by the computation’s order of commitment; that is, the order in which instructions are removed from the processor queues. Let us also note that a response can be generated only if its instruction is removed from the processor queue, hence committed. Therefore, in the logical order, a response \( r \) will follow all the responses that correspond to instructions which are committed before the instruction corresponding to \( r \). The \( SC(j, k) \) machine can at most retain \( k \cdot p - 1 \) many such committed instructions. This immediately implies that \( f(i) = i + k \cdot p - 1 \).

This result should not come as a surprise as the \( SC \) machines are implementing a generalized version of linearizability where out of order completion (response generation) and more than one pending instruction per processor are both possible. However, due to its semantics, an \( SC \) machine does not realize a unary relation but a binary relation instead. That is, the language of an \( SC \) machine is not a subset of \( \Sigma^* \) but that of \( (\Sigma^2)^* \), where \( \Sigma \) is the set of instructions and responses. In this sense, linearizability is not necessarily as restrictive.

This point might need more elaboration. Consider the following computation, where instructions and responses are given in the same string:

\[
!a_1 \ a_1^1! \ a_2^2 \ a_1^1 \ ?a_2 \ a_2^1?
\]

This computation cannot be generated by a linearizable system. It cannot be generated by an \( SC \) machine either. On the other hand, it is easy to see that it can be generated by a sequentially consistent machine.
However, consider the following computation which can be generated by $SC(2, 3)$:

$!a_1^1 \! !a_2^2 \ ?a_2 \ a_1^1! \ a_2^2! \ a_2^1?$

According to the semantics given in [3], the two computations are equivalent because they agree on the order of instructions and on the order of responses, even though their temporal executions are not the same. Since $SC$ machines are transducers, the previous program/execution pair corresponding to a non-linearizable computation is included in the language of the above $SC$ machine.

Therefore, we claim that the use of $SC$ machines as an alternative definition for sequential consistency is an improvement over the use of linearizability.

In [3], we have demonstrated the use of this approach while we were proving the sequential consistency of the lazy caching algorithm [5]. Lazy caching algorithm is a non-trivial implementation of a sequentially consistent system which is non-linearizable. Due to the possibility of delaying certain operations for an unbounded number of transitions, it permits certain computations which should be prohibited intuitively. Consider the following computation:

$!a_1^1 \ a_1^1! \ ?a_2 \ a_2^0? \ ?a_1 \ a_1^1? \ ?a_2 \ a_2^0? \ ?a_1 \ a_1^1? \ldots$

This computation where the second processor never updates its cache and keeps reading the initial value 0 can be generated by a lazy caching implementation. That implies that the lazy caching implementations are not logically bounded; unbounded precisely due to computations of the above kind. It is normal then to observe that there does not exist an $SC$ machine whose language contains all possible program/execution pairs generated by a lazy caching implementation (see [3]). However, we have proved the sequential consistency of a fair implementation of lazy caching by proving that for any such
implementation there is an $SC$ machine whose language contains that of the implementation. So, we believe that even though lazy caching is sequentially consistent, it needs some notion of fairness\(^3\).

We should add that we have also proved that the (infinite) union of the languages of all the $SC$ machines is exactly equal to the set of all program/execution pairs that can be generated by sequentially consistent systems. Besides providing one with an infinite state machine whose language is equal to this union, to prove the sequential consistency of an (unfair) lazy caching implementation, this equality shows that the $SC$ machines do not define an ad-hoc subset of sequential consistency while relating temporal execution to logical order for each computation.

6 Further Remarks

We have so far not alluded to one major benefit of using the $SC$ machines. Since these machines enable the verification of sequential consistency to be formulated as a regular language inclusion problem (see [3]), they have the additional and much desirable property of making the checking process fully automatic. So not only do we get a better characterization of what is intuitively correct, but also we obtain an algorithm for its verification.

Another possible way to restrict the definition of sequential consistency is to designate language inclusion as a necessary condition. That is, that the language generated by the shared memory system is contained in the set of all sequentially consistent computations will not imply the sequential consis-

\(^3\) For a detailed analysis, see [3].
tency of the system. Sufficiency then can be proved by checking certain state formulas whose truth depends on the transition structure of the finite state structure. For instance, temporal logic formulas can be checked to ensure that a state where all processors have the same data value for each address is always inevitable.

We should note that we do not favor this approach. Instead of dealing with a regular language inclusion problem as in the case of SC machines, this two-step method requires the verification personnel to learn the memory system at a lower level of abstraction. This is due to the fact that the second step (sufficiency) might require certain assertions about states. One example was given above: what exactly is meant by “all processors have the same data value for each address”?

Finally, we should note that this problem can easily be adapted to coherence as well. The synchronizing sequence would be defined per address; the mischievous machine this time will not communicate the updated data values for an address if the first instruction to that address by each processor is a write instruction. And a possible remedy would be again given by a family of finite-state transducers each of which is implementing coherence, much like the SC machines.

References


