Toward A Decidable Notion of Sequential Consistency *

Jesse D. Bingham
Department of Computer Science
University of British Columbia
jbingham@cs.ubc.ca

Anne Condon
Department of Computer Science
University of British Columbia
condon@cs.ubc.ca

Alan J. Hu
Department of Computer Science
University of British Columbia
ajh@cs.ubc.ca

Categories and Subject Descriptors

B.3.3 [**Performance Analysis and Design Aids**]: [Formal Models]; C.0 [**Computer Systems Organization**]: General—*systems specification methodology*; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: [Multiple-instruction-stream, multiple-data-stream processors (MIMD)]

General Terms

Theory,Verification

Keywords

sequential consistency, shared memory systems, memory model

ABSTRACT

A memory model specifies a correctness requirement for a distributed shared memory protocol. Sequential consistency (SC) is the most widely researched model; previous work [1] has shown that, in general, the SC verification problem is undecidable. We identify two aspects of the formulation found in [1] that we consider to be highly unnatural; we call these non-prefix-closedness and prophetic inheritance. We conjecture that preclusion of such behavior yields a decidable version of SC, which we call decisive sequential consistency (DSC). We also introduce a structure called a *view window* (VW), which retains information about a protocol's history, and we define the notion of a *VW-bound*, which essentially bounds the size of the VWs needed to maintain DSC. We prove that the class of DSC protocols with VW-bound k is decidable; left conjectured is the hypothesis that all DSC protocols have such a bound, and further that the bound is computable from the protocol description. This hypothesis is true for all real protocols known to us; we verify its truth for the Lazy Caching protocol [2].

1. INTRODUCTION

A *distributed shared-memory protocol* (or, simply *protocol*) is an algorithmic and architectural description of how multiple CPUs share a common memory space. A *memory model* defines a set of allowable behaviors for a protocol, and formally specifies how the memory system will appear to the programmer [3]. Hence the correctness of a protocol is always with respect to a specific memory model.

Much research has be conducted on the shared memory model *sequential consistency (SC)* since its inception [4]. A large portion of this research has pertained to the formal verification of this property. Verification techniques have been proposed that range on the automation axis from being manual hand-proof methodologies [5, 6], to fully automatic procedures [7, 8], and many points between [9, 10, 11]. Here we focus on fully automatic verification of SC, and reserve the term *verification* to refer to such.

The seminal definition of SC [4] was informal. The subsequent literature bore several non-equivalent formalizations. Most consider only finite behaviors, though recent work has examined the ramification of interpreting SC over infinite runs [12]. An important disagreement actually arises from how the notion of *protocol* is modeled. Some works take a protocol as *any* regular set over the alphabet of memory actions $\mathcal{M}$, while others further require that the set be prefix-closed. We call these models *not necessarily prefix-closed (NNPC)* and *prefix-closed (PC)*.

Alur et al. have shown that SC under the NNPC protocol model is undecidable [1]. This formulation admits a protocol behavior, which we entitle *prophetic inheritance*, in which a read event receives a data value from a write event that occurs in the former's temporal future. The proof of the undecidability theorem depends upon the allowance of prophetic inheritance.

We believe that the PC protocol model more tightly fits the realm of real protocols. Indeed, protocol designers typically describe the protocol as an automaton where all reachable states are implicitly accepting [13]. We also believe that a prudent formalization of SC should disallow prophetic inheritance, since a protocol that supports this oddity must either predict the future with 100% accuracy, or govern the value a processor may write. Both of these features fall well outside of what a protocol can and should do[1].

*This work was supported in part by a research grant and a graduate fellowship from the Natural Science and Engineering Research Council of Canada.

[1]At first glance, prohibiting prophetic reads might appear to preclude the recently proposed optimization of *load value prediction*[14]. In reality, however, load value prediction does not rely on prophetic reads. In the proposed schemes, the predicted value is remembered from previous reads of the value, which ultimately came from an earlier write. Furthermore, to guarantee correctness, the schemes either enforce that the predicted value is coherent with the memory system and is therefore not prophetic, or else later verify that the predicted value was correct, quashing any mis-predicted instructions, in which case the read effectively doesn't occur until after the prediction has been verified.

SPAA'03, June 7–9, 2003, San Diego, California, USA.

The primary contributions of this paper are as follows. A new version of SC, called decisive sequential consistency (DSC), is defined. Informally, DSC requires that when a memory read event occurs, the protocol "knows" which write event the data value stemmed from, and the protocol never needs to "change its mind" in the future in order to maintain SC. It is shown that DSC is equivalent to *past-time sequential consistency* (PTSC), the latter being SC with the restriction that no read event inherits its data from a write that has yet to occur. All SC protocols with which we are familiar actually implement the slightly stronger DSC. We then define an object called a *view window* (VW). A VW is essentially a compressed representation of a memory trace. We define subclasses of DSC that are bounded in the size of the VWs needed to maintain DSC, and show that these classes are decidable. For illustration of the power of this approach, we show that Lazy Caching [2] is in such a class. It is our intuition that any DSC protocol *must* implicitly have a finite set of VWs associated with any state; these VWs describe what has been "remembered" by the protocol regarding its execution history, and are utilized to maintain DSC in the future. We formalize this intuition in a conjecture. If our conjecture holds, it follows that DSC itself is decidable.

2. DEFINITIONS

For a natural n, let $\mathbb{N}_n = \{1,\ldots,n\}$. Given an alphabet Σ, a *sequence* or *string* over Σ is a mapping $\tau : \mathbb{N}_\ell \to \Sigma$ for some ℓ, and we define $|\tau| = \ell$. We sometimes refer to $i \in \mathbb{N}_\ell$ as an *index* (of τ). A sequence can be expressed by writing its symbols in order, i.e. $\tau(1),\ldots,\tau(|\tau|)$. Hence the use of the terms *left* and *right* to refer to features of the sequence makes sense, we sometimes use the term *last* to mean rightmost. The prefix of τ of length i is denoted $\tau[i]$. Given a sequence τ and a set A, the *projection* of τ onto A is the sequence $\tau\!\uparrow\!A$ obtained by deleting all symbols not in A. For $\alpha \in \Sigma$ we say that a string $\tau' \in \Sigma^*$ is obtained from τ by *inserting* α *at position* j if $\tau' = \tau(1)\ldots\tau(j)\alpha\tau(j+1)\ldots\tau(|\tau|)$, where j may be any element of $\mathbb{N}_{|\tau|} \cup \{0\}$

The set of *memory actions* is the set $\mathcal{M}(P,B,V) = \{R,W\} \times \mathbb{N}_P \times \mathbb{N}_B \times \mathbb{N}_V$. Intuitively, P, B, and V are respectively the number of processors, addresses, and data values under consideration. For the remainder of this article, we fix P, B, and V and abbreviate $\mathcal{M} = \mathcal{M}(P,B,V)$. For $\alpha = (o,p,b,v) \in \mathcal{M}$, we define $op(\alpha) = o$, $proc(\alpha) = p$, $addr(\alpha) = b$, and $val(\alpha) = v$. We call a string over $\mathcal{M}$ a *memory trace* (or simply *trace*), and the sets $\mathbb{N}_P$, $\mathbb{N}_B$, and $\mathbb{N}_V$ respectively the *processors*, *addresses*, and *values*. We also define subsets of $\mathcal{M}$ using the wild-card symbol $*$. For instance, $(*,p,b,*)$ is the set $\{\alpha \in \mathcal{M} \mid proc(\alpha) = p \wedge addr(\alpha) = b\}$. The set $(R,*,*,*)$ constitutes the *read actions* while the elements of $(W,*,*,*)$ are the *write actions*.

Given a memory trace τ, an index i, and address b, we define $lw(\tau,i,b)$ to be the greatest index $j \le i$ such that $op(\tau(j)) \in (W,*,b,*)$ or to be 0 if no such j exists. Intuitively, $lw(\tau,i,b)$ is the index of the last write to address b that is not greater than i. A memory trace σ is said to be *serial* if, for all indices i, we have $op(i) = R$ implies both $lw(\sigma,i,addr(\sigma(i))) \neq 0$ and $val(\sigma(i)) = val(\sigma(lw(\sigma,i,addr(\sigma(i)))))$. In other words, in a serial trace each read action returns the value of the last write action to the same address[2]. As a notational convenience, we define $val(\sigma(0))$ to be a distinguished constant $\perp$, which signifies that no value has been assigned. Given trace τ and processor p, the index of the last action in $(*,p,*,*)$ is extracted via $lpa(\tau,p) = \max(\{i \mid proc(\tau(i)) = p\} \cup \{0\})$. Here, the acronym lpa stands for *last processor action*.

We typically use σ to represent a serial trace, while τ represents a trace that is not necessarily serial.

A *reordering* of a trace τ is a permutation π on $\mathbb{N}_{|\tau|}$, and we define the trace τ^π to be $\tau(\pi^{-1}(1))\ldots\tau(\pi^{-1}(|\tau|))$. We allow applying a reordering π to a prefix of length $k < |\tau|$ by defining the permutation $\pi' : \mathbb{N}_k \to \mathbb{N}_k$ such that $\pi'(i) < \pi'(j) \Leftrightarrow \pi(i) < \pi(j)$ for all $i,j \in \mathbb{N}_k$, and then defining the reordered prefix $\tau[k]^\pi$ to be equal to $\tau[k]^{\pi'}$. Associated with a trace is a partial order which places actions of the same processor in the order they occur in the trace; the resulting relation is called the *processor order*. Formally, given a trace τ we define the partial order $<^{po}_\tau$ on the indices such that $i <^{po}_\tau j$ iff $(proc(i) = proc(j)) \wedge (i < j)$. A *serial reordering* of τ is a reordering π such that τ^π is serial, and further π respects $<^{po}_\tau$, i.e. $i <^{po}_\tau j$ iff $\pi(i) < \pi(j)$, or equivalently, for all processors p, we have $\tau\!\uparrow\!(*,p,*,*) = \tau^\pi\!\uparrow\!(*,p,*,*)$. A trace is said to be *sequentially consistent (SC)* if it has a serial reordering. If τ is SC and has serial reordering π, then we define an *inheritance relation* $\mapsto^\pi_\tau$ on the indices of τ by $i \mapsto^\pi_\tau j$ if $op(\tau(i)) = W$, $op(\tau(j)) = R$, $addr(\tau(i)) = addr(\tau(j))$, and $\pi(i) = lw(\tau^\pi, \pi(j), addr(\tau(j)))$. When $i \mapsto^\pi_\tau j$ and τ and π are understood from context, we say j *inherits from* i. If a trace σ is serial, then we use $\mapsto_\sigma$ to denote $\mapsto^{\mathrm{id}}_\sigma$, where id is the identity permutation.

Let σ be a serial trace. For any i such that $op(\sigma(i)) = W$, we define the *last read of* i by $lr(\sigma,i) = \max(\{i' \mid i \mapsto_\sigma i'\} \cup \{i\})$. Intuitively, $lr(\sigma,i)$ provides the *last read* that inherits from i. For any $j \in \{0,\ldots,|\sigma|\}$ and address b, we define the *inheritance range* predicate $IR(\sigma,j,b)$ that is true iff $lw(\sigma,j,b) \neq 0$ and $j < lr(\sigma, lw(\sigma,j,b))$. Thus $IR(\sigma,j,b)$ holds whenever position j is between a write to address b and a read that inherits from the write.

A *protocol* $\mathcal{P}$ is a 4-tuple $(S(\mathcal{P}), \mathcal{A}(\mathcal{P}), \delta(P), I(\mathcal{P}))$. $S(\mathcal{P})$ is the set of *states* and must be finite. $\mathcal{A}(\mathcal{P})$ is the set of protocol *actions*, where we require $\mathcal{M} \subseteq \mathcal{A}(\mathcal{P})$. The *transition relation* $\delta(\mathcal{P})$ is a subset of $S(\mathcal{P}) \times \mathcal{A}(\mathcal{P}) \times S(\mathcal{P})$. Let $\delta^*(\mathcal{P}) \subseteq S(\mathcal{P}) \times \mathcal{A}(\mathcal{P})^* \times S(\mathcal{P})$ be the natural generalization of $\delta(\mathcal{P})$ to strings: $(s_0, \tau, s_{|\tau|}) \in \delta^*(\mathcal{P})$ if and only if there exists a sequence of states $s_1, s_2, \ldots, s_{|\tau|-1}$ in $S(\mathcal{P})$ such that for all i, $1 \le i < |\tau|$, $(s_{i-1}, \tau(i), s_i) \in \delta(\mathcal{P})$. The set $I(\mathcal{P}) \subseteq S(\mathcal{P})$ are the *initial states*. A *run* of $\mathcal{P}$ is a sequence $r = a_1,\ldots,a_\ell$ over $\mathcal{A}$ such that there exists states $s_1,\ldots,s_{\ell+1} \in S(\mathcal{P})$ with $s_1 \in I(\mathcal{P})$ and $(s_i, a_i, s_{i+1}) \in \delta(\mathcal{P})$ for all $i \in \mathbb{N}_\ell$. The *trace* of a run r is the sequence $trace(r) = r\!\uparrow\!\mathcal{M}$. Given a protocol $\mathcal{P}$, the *trace set* of $\mathcal{P}$ is the set $traces(\mathcal{P}) = \{trace(r) \mid r \text{ is a run of } \mathcal{P}\}$. A protocol is characterized as *serial* if all of its traces are *serial*; similarly, a protocol is SC if all of its traces are SC.

3. DECISIVE SEQUENTIAL CONSISTENCY

In this section, we define two SC variants, DSC and PTSC, and we prove their equivalence.

Definition 1 (Decisive SC) *A trace* τ *is said to be* decisive sequentially consistent (DSC) *if there exists a serial reordering* π *of* τ *such that for any prefix* $\tau[\ell]$ *and* $1 \le i,j \le \ell$ *we have: (1)* $\tau[\ell]^\pi$ *is serial, and (2)* $i \mapsto^\pi_{\tau[\ell]} j \Leftrightarrow i \mapsto^\pi_\tau j$. *In this case we say that* π *is a* DSC-reordering *of* τ.

A protocol is DSC if all of its traces are DSC and we denote the set of all such protocols by *DSC*. On an intuitive level, a DSC protocol is a SC protocol that never needs to "change its mind"

[2] For simplicity's sake, we have chosen to disallow initial values; this approach is also taken in [15]. It should be clear that all theory herein can be generalized to handle formalizations that allow reading of initial data values.

regarding which write wrote-to a given read. A trace being DSC implies that all prefixes of the trace are SC (moreover they are all DSC). However, having all prefixes SC is insufficient for DSC, as this example trace ρ demonstrates. To aid comprehension, we attach a number underneath each memory action giving its position in ρ.

$$\rho = \underset{1}{(W,2,1,2)}\underset{2}{(W,2,1,1)}\underset{3}{(W,2,2,1)}\underset{4}{(R,1,2,1)}\underset{5}{(W,3,2,1)}\underset{6}{(R,1,1,2)}$$

A serial reordering π exists:

$$\rho^{\pi} = \underset{5}{(W,3,2,1)}\underset{4}{(R,1,2,1)}\underset{1}{(W,2,1,2)}\underset{6}{(R,1,1,2)}\underset{2}{(W,2,1,1)}\underset{3}{(W,2,2,1)}$$

It can be shown that *any* serial reordering must have 4 inherit from 5; this follows from the fact that any serial reordering must order 6 prior to 2. However, restricting such a reordering to the prefix τ of length 4 will not give a serial reordering. Continuing our example, we find

$$\rho[4]^{\pi} = \underset{4}{(R,1,2,1)}\underset{1}{(W,2,1,2)}\underset{2}{(W,2,1,1)}\underset{3}{(W,2,2,1)}$$

which is clearly not serial. The reader may confirm that all prefixes of τ are SC, for instance $\rho[4]$ may be serially reordered as

$$\underset{1}{(W,2,1,2)}\underset{2}{(W,2,1,1)}\underset{3}{(W,2,2,1)}\underset{4}{(R,1,2,1)}$$

We now present a seemingly different SC variant, PTSC, and prove that it is equivalent to DSC. Our motivation for presenting DSC in the guise of PTSC is to provide greater intuition regarding what type of behaviors these models allow and disallow.

Definition 2 (past-time SC) *A trace τ is said to be* past-time sequentially consistent (PTSC) *if there exists a serial reordering π of τ such that $i \mapsto^{\pi}_{\tau} j$ implies $i < j$. In this case we say that π is a* PTSC-reordering *of τ.*

Theorem 1 *π is a DSC-reordering of τ iff π is a PTSC-reordering of τ.*

Proof: ($\Rightarrow$) Suppose there exist i and j such that $i \mapsto^{\pi}_{\tau} j$ and $i > j$. Consider the prefix $\tau[i-1]$. Clearly $i \not\mapsto^{\pi}_{\tau[i-1]} j$, since i isn't an index of $\tau[i-1]$. ($\Leftarrow$) Now assume π is a PTSC-reordering of τ. We prove by (decreasing) induction on ℓ that properties 1 and 2 from definition 1 hold w.r.t $\tau[\ell]$. For $\ell = |\tau|$ we have $\tau[\ell] = \tau$, hence the properties hold trivially. Now assume that the properties hold in $\tau[\ell]$. If $\tau(\ell)$ is a write, then there does not exist i such that $\ell \mapsto^{\pi}_{\tau[\ell]} i$, thus $\tau[\ell-1]^{\pi}$ satisfies both properties. Now if $\tau(\ell)$ is a read, then $\tau[\ell-1]^{\pi}$ is trivially serial, since removing a read from a serial trace will always yield a serial trace; property 2 is also satisfied given the inductive hypothesis. ∎

4. VIEW WINDOWS

Ideally, we would like to construct a protocol whose trace set is exactly the set of all DSC memory traces, which would allow automatic verification of protocols. In section 5, we partly realize this goal. In this section, we introduce concepts and notation needed to describe the protocol of section 5.

As motivation, we first describe informally an infinite-state protocol, $\mathcal{G}_{\infty}$, whose trace set is the set of all DSC memory traces, and then consider how the state space of $\mathcal{G}_{\infty}$ could be made finite. The states of $\mathcal{G}_{\infty}$ are simply the serial memory traces, with the initial state being the empty trace, which we refer to as σ_0. The set of transitions $\delta(\mathcal{G}_{\infty})$ contains all triples $(\sigma, \alpha, \sigma')$ where α is a memory action, and σ' is a serial trace obtained by inserting α at some position j in σ, where $j \geq lpa(\sigma, proc(\alpha))$. Suppose we write $\sigma_0 \rightarrow_{\tau} \sigma$ if $(\sigma_0, \tau, \sigma) \in \delta^*(\mathcal{G}_{\infty})$. Then, it can be shown by induction on $|\tau|$ that

Claim 1 *For all traces τ, $\sigma_0 \rightarrow_{\tau} \sigma$ if and only if τ is DSC and $\sigma = \tau^{\pi}$ for some DSC reordering π of σ.*

Thus, the trace set of the protocol $\mathcal{G}_{\infty}$ is exactly the set of DSC traces.

Intuitively, to obtain a (restricted) finite state version of $\mathcal{G}_{\infty}$, it is necessary to avoid storing all of a serial trace σ in the state. The view window presented in section 4.1 is a condensed version of a serial trace, developed for this purpose. View windows generalize the concept of *window* found in [10]. We also need operations to update view windows in a manner consistent with the trace produced by the protocol. The needed operations and assertions of their correctness are presented in section 4.2.

4.1 View Window Defined

A view window is an abstraction of a serial trace σ that can be used to determine, for a few positions j and any memory action α, whether the string σ' obtained from σ by inserting α at position j is serial. Accordingly, for certain positions j, a view window contains a *view* (of position j). For each address b the view contains the following information:

- The value of the last write operation to b before position j of σ. This can be used to determine what is the acceptable value of a read operation to address b, inserted at position j.
- A pair of tags. One tag indicates whether or not the predicate $IR(\sigma, j, b)$ holds; that is, position j is between a write to address b in σ and a read that inherits from the write. This tag can be used to determine whether the protocol is free to insert a write operation to address b at position j. The tag has value F (**f**ree to write) or O (reads **o**nly). The other tag is needed to help update the F/O tag. It has value L (**l**ast) or N (**n**ot last).

In addition to views, the view window contains additional information, called the lp function, which is useful in determining whether insertion of α respects processor order.

We now formally define view windows. An example of a serial trace σ and a corresponding view window is given in parts (a), (b) of Table 1. Define the set of *tagged values* $T = (\mathbb{N}_V \cup \{\perp\}) \times \{L, N\} \times \{O, F\}$. Given a tagged value $z = (v, t_1, t_2)$, we extract the components via $val(z) = v$, $tag_1(z) = t_1$, and $tag_2(z) = t_2$. A *view* is a function $\mathsf{v}\colon \mathbb{N}_B \rightarrow T$. Given a serial string $\sigma \in \mathcal{M}^*$, a subsequence ps of $0, \ldots, |\sigma|$ such that ps includes $|\sigma|$ is called a *position sequence (for σ).*

Definition 3 (VW-set,VW) *Given σ and an accompanying position sequence ps, we define the VW-set $VW(\sigma, \mathsf{ps})$ to be the set of all pairs $(\mathsf{v}, \mathsf{lp})$ that satisfy the following: v is a sequence of views with $|\mathsf{v}| = |\mathsf{ps}|$ defined as follows.*

1. $val(\mathsf{v}(i)(b)) = val(\sigma(lw(\sigma, \mathsf{ps}(i), b)))$
2. $tag_1(\mathsf{v}(i)(b))$

$$= \begin{cases} L & \text{if } i = 1 \vee [i > 1 \wedge \mathsf{ps}(i-1) < lw(\sigma, \mathsf{ps}(i), b)] \\ N & \text{otherwise} \end{cases}$$

3.

$$tag_2(\mathsf{v}(i)(b)) = \begin{cases} O & \text{if } IR(\sigma, \mathsf{ps}(i), b) \\ F & \text{otherwise} \end{cases}$$

and lp *is a function* $\mathbb{N}_P \to \mathbb{N}_{|\mathsf{v}|}$ *which must satisfy*

4. $\mathsf{ps}(\mathsf{lp}(p)) \geq lpa(\sigma, p)$ *for all* $p \in \mathbb{N}_P$.

A pair $w = (\mathsf{v}, \mathsf{lp})$ *is called a* view window (VW) *(of* σ*) if there exists* ps *such that* $(\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$. *The* size *of a VW* $(\mathsf{v}, \mathsf{lp})$ *is* $|(\mathsf{v}, \mathsf{lp})| = |\mathsf{v}|$.

4.2 Operations on View Windows

Recall that in the infinite-state protocol $\mathcal{G}_\infty$, upon each transition, the state (a serial trace) is updated by insertion of a memory action. In a protocol that abstracts states as view windows, we need operations (state transitions) that update view windows upon insertion of a memory action. We define five such operations, or functions that take VWs to VWs. The *delete* function removes a view from a view window (in order to keep the view small). The *insert* function inserts a view into a view window (corresponding to the insertion of a memory action α into the serial trace that the view window abstracts). The *hop* function makes it possible to advance $\mathsf{lp}(p)$ for any processor p (in order that an operation of processor p is inserted after the appropriate view). Finally, the *unfree* and *bind* operations update the view tags appropriately. Example applications of these functions are given in Table 1.

We now formally define the five operations. In all descriptions, $(\mathsf{v}', \mathsf{lp}')$ is the VW returned by the function, and p and b may be any processor and address, respectively.

- $delete((\mathsf{v}, \mathsf{lp}), h)$ requires $h \in \mathbb{N}_{|\mathsf{v}|-1}$. The new view v' is defined according to

$$\mathsf{v}'(j)(b) = \begin{cases} \mathsf{v}(j)(b) & \text{if } j < h \\ (val(\mathsf{v}(h+1)(b)), L, tag_2(\mathsf{v}(h+1)(b))) & \\ & \text{if } j = h \\ & \quad \wedge tag_1(\mathsf{v}(h)(b)) = L \\ & \quad \wedge tag_1(\mathsf{v}(h+1)(b)) = N \\ \mathsf{v}(j+1)(b) & \text{otherwise} \end{cases}$$

  Finally, for all $p \in \mathbb{N}_P$ we have $\mathsf{lp}'(p) = \mathsf{lp}(p)$ if $\mathsf{lp}(p) \leq h$, and $\mathsf{lp}'(p) = \mathsf{lp}(p) - 1$ otherwise.

- $hop((\mathsf{v}, \mathsf{lp}), p, h)$ requires $\mathsf{lp}(p) < h \leq |\mathsf{v}|$. *hop* leaves v unchanged, i.e. $\mathsf{v}' = \mathsf{v}$. lp' is everywhere equal to lp with the exception $\mathsf{lp}'(p) = h$.

- $insert((\mathsf{v}, \mathsf{lp}), p, \nu)$ is called against a view ν. Intuitively, v' is obtained from v by inserting ν at position $\mathsf{lp}(p)$. lp' is defined by $\mathsf{lp}'(q) = \mathsf{lp}(q)$ if $\mathsf{lp}(q) \leq \mathsf{lp}(p) \wedge p \neq q$; otherwise $\mathsf{lp}'(q) = \mathsf{lp}(p) + 1$.

- $unfree((\mathsf{v}, \mathsf{lp}), p, b)$: Let[3]

$$j = \max(\{i \mid i < \mathsf{lp}(p) \wedge tag_1(\mathsf{v}(i)(b)) = L\}).$$

  Then v' is everywhere equal to v, with the possible exceptions: for all $k \in \{j, \ldots, \mathsf{lp}(p) - 1\}$ we have $\mathsf{v}'(k)(b) = (val(\mathsf{v}(k)(b)), tag_1(\mathsf{v}(k)(b)), O)$. Finally, lp' is simply equal to lp.

- $bind((\mathsf{v}, \mathsf{lp}), p, b, v)$: Here we require $v \in \mathbb{N}_V$. This function leaves lp unchanged, and if $\mathsf{lp}(p) = |\mathsf{v}|$, then $\mathsf{v}' = \mathsf{v}$. Otherwise, let j be minimal such that $j > \mathsf{lp}(p)$ and $tag_1(\mathsf{v}(j)(b)) = L$; if no such j exists take $j = |\mathsf{v}| + 1$. Then v' is the same as v, with the exception that for all $k \in \{\mathsf{lp}(p) + 1, \ldots, j - 1\}$ we have $\mathsf{v}'(k)(b) = (v, N, F)$.

We define the partial order $\leq_{vw}$ on VWs such that $x \leq_{vw} y$ iff y can be obtained from x by 0 or more *delete* and *hop* operations.

Just as in our protocol $\mathcal{G}_\infty$, where the protocol "evolves" on memory actions from serial trace to serial trace, we can also define what it means to evolve on a memory action from view window to view window.

Definition 4 *Given VWs* $w = (\mathsf{v}, \mathsf{lp})$ *and* w' *and* $\alpha \in \mathcal{M}$*, we say that* w can directly α-evolve to w' *if the following conditions are satisfied.*

1. *If* $\alpha = (R, p, b, v)$*, then we require* $v = val(\mathsf{v}(\mathsf{lp}(p))(b)) \neq \perp$*, and the following equation must hold:*

$$w' = unfree(insert(w, p, \nu), p, b)$$

   where ν *is given by* $\nu(a) = (val(\mathsf{v}(\mathsf{lp}(p))(a)), N, tag_2(\mathsf{v}(\mathsf{lp}(p))(a)))$ *for each* $a \in \mathbb{N}_B$.

2. *If* $\alpha = (W, p, b, v)$*, then we must have*

   (a) $tag_2(\mathsf{v}(\mathsf{lp}(p))(b)) = F$

   (b) $w' = bind(insert(w, p, \nu), p, b, v)$*, where* ν *is given by*

$$\nu(a) = \begin{cases} (v, L, F) & \text{if } a = b \\ (val(\mathsf{v}(\mathsf{lp}(p))(a)), N, tag_2(\mathsf{v}(\mathsf{lp}(p))(a))) & \\ & \text{otherwise} \end{cases}$$

If there exists w' *such that* w *directly* α*-evolves to* w'*, we say that* w *is* directly α-enabled. *If there exist* $w_1 = (\mathsf{v}_1, \mathsf{lp}_1)$ *and* $w_2 = (\mathsf{v}_2, \mathsf{lp}_2)$ *such that* $w \leq_{vw} w_1$ *and* $w_2 \leq_{vw} w'$*, and* w_1 *directly* α*-evolves to* w_2*, we say* w α-evolves to w'*, denoted* $w \rightsquigarrow_\alpha w'$*. If there exists* w' *such that* $w \rightsquigarrow_\alpha w'$ *we say that* w *is* α-enabled.

This completes our description of operations on view windows and the associated α-evolves relation. Before proving properties of this relation in the rest of this section, we provide some informal intuition. First we note that we can generalize the notion of α-evolves to sequences τ of memory operations, where $w \rightsquigarrow_\tau w'$ if and only if there is a sequence $w_1, w_2, \ldots w_{|\tau|-1}$ of view windows such that $w \rightsquigarrow_{\tau(1)} w_1$, $w_{i-1} \rightsquigarrow_{\tau(i)} w_i$ for $1 < i \leq |\tau| - 1$, and $w_{\tau(|\tau|-1)} \rightsquigarrow_{\tau(|\tau|)} w'$. Let w_0 be the view window of the empty trace, that is, $w_0 = (\mathsf{v}_0, \mathsf{lp}_0)$ where v_0 is a singleton view sequence with $\mathsf{v}_0(b) = (\perp, L, F)$ for each address b and $\mathsf{lp}_0(p) = 1$ for all $p \in \mathbb{N}_P$. It turns out that

Claim 2 *For all traces* τ*,* $w_0 \rightsquigarrow_\tau w$ *if and only if* τ *is DSC and* w *is a view window of* τ^π *for some DSC reordering* π *of* τ.

Claim 2 is analogous to Claim 1 for serial traces given at the start of section 4, except the former pertains to view windows. Roughly, Claim 2 can be proved by induction on $|\tau|$, using Theorem 2 which is given at the end of this section. The next three lemmas build up to the proof of Theorem 2. We don't actually prove Claim 2, but rather prove a variant in section 5, in which the sizes of view windows are bounded.

[3] *unfree* is only used when $\mathsf{lp}(p) > 1$ and def. 3.2 ensures that $tag_1(\mathsf{v}(1)(b)) = L$, hence j is well-defined.

(a)	σ		$(W,2,1,2)$		$(W,1,1,1)(W,1,2,1)(R,1,1,1)$		$(R,1,2,1)$				$(W,2,2,2)(R,2,1,1)$		$(R,1,2,2)$	
(b)	j	1		2		3		4				5		6
	$\mathsf{ps}(j)$	0		1		4		5				7		8
	$\mathsf{v}(j)$	$\begin{bmatrix}(\perp,LF)\\(\perp,LF)\end{bmatrix}$		$\begin{bmatrix}(2,LF)\\\perp\end{bmatrix}$		$\begin{bmatrix}(1,LO)\\(1,LO)\end{bmatrix}$		$\begin{bmatrix}(1,NO)\\(1,NF)\end{bmatrix}$				$\begin{bmatrix}(1,NF)\\(2,LO)\end{bmatrix}$		$\begin{bmatrix}(1,NF)\\(2,NF)\end{bmatrix}$
	$\mathsf{lp}^{-1}(j)$	$\emptyset$		$\{3\}$		$\emptyset$		$\emptyset$				$\{2\}$		$\{1\}$
(c)	j	1		2		3		4						5
	$\mathsf{ps}_1(j)$	0		1		4		5						8
	$\mathsf{v}_1(j)$	$\begin{bmatrix}(\perp,LF)\\(\perp,LF)\end{bmatrix}$		$\begin{bmatrix}(2,LF)\\\perp\end{bmatrix}$		$\begin{bmatrix}(1,LO)\\(1,LO)\end{bmatrix}$		$\begin{bmatrix}(1,NO)\\(1,NF)\end{bmatrix}$						$\begin{bmatrix}(1,NF)\\(2,LF)\end{bmatrix}$
	$\mathsf{lp}_1^{-1}(j)$	$\emptyset$		$\emptyset$		$\emptyset$		$\{3\}$						$\{1,2\}$
(d)	σ_2		$(W,2,1,2)$		$(W,1,1,1)(W,1,2,1)(R,1,1,1)$		$(R,1,2,1)$		$(W,3,2,4)$		$(W,2,2,2)(R,2,1,1)$		$(R,1,2,2)$	
(e)	j	1		2		3		4		5				6
	$\mathsf{ps}_2(j)$	0		1		4		5		6				9
	$\mathsf{v}_2(j)$	$\begin{bmatrix}(\perp,LF)\\(\perp,LF)\end{bmatrix}$		$\begin{bmatrix}(2,LF)\\\perp\end{bmatrix}$		$\begin{bmatrix}(1,LO)\\(1,LO)\end{bmatrix}$		$\begin{bmatrix}(1,NO)\\(1,NF)\end{bmatrix}$		$\begin{bmatrix}(1,NO)\\(4,LF)\end{bmatrix}$				$\begin{bmatrix}(1,NF)\\(2,LF)\end{bmatrix}$
	$\mathsf{lp}_2^{-1}(j)$	$\emptyset$		$\emptyset$		$\emptyset$		$\emptyset$		$\{3\}$				$\{1,2\}$

Table 1: VW and VW operation examples, note that a view ν is expressed as a column vector $[\mathsf{v}(1)\ \ \mathsf{v}(2)]^T$: (a) a serial string $\sigma \in \mathcal{M}(3,2,2)$, (b) a position sequence ps for σ and a VW $w = (\mathsf{v},\mathsf{lp}) \in VW(\sigma,\mathsf{ps})$, (c) the VW $w_1 = (\mathsf{v}_1,\mathsf{lp}_1) = hop(delete(w,5),3,4)$, along with position sequence ps_1 such that $w_1 \in VW(\sigma,\mathsf{ps}_1)$, (d) a serial string σ_2 obtained by performing an insertion of $(W,3,2,4)$ against σ, (e) a position sequence ps_2 for σ_2 and a VW $w_2 = (\mathsf{v}_2,\mathsf{lp}_2) \in VW(\sigma_2,\mathsf{ps}_2)$, where w_1 directly $(W,3,2,4)$-evolves to w_2. Some other points of note here are: $w \rightsquigarrow_{(W,3,2,4)} w_2$ and $w \leq_{vw} w_1$. Also, the set of all α such that w_1 is directly α-enabled is $(W,1,*,*) \cup (W,2,*,*) \cup (W,3,2,*) \cup \{(R,1,1,1),(R,1,2,2),(R,2,1,1),(R,2,2,2),(R,3,1,1),(R,3,2,1)\}$.

v_0	$(\perp,LF)$								
lp_0^{-1}	$\{1,2\}$								
$\tau[1]^\pi$		$(W,1,1,1)$							
v_1	$(\perp,LF)$		$(1,LF)$						
lp_1^{-1}	$\{2\}$		$\{1\}$						
$\tau[2]^\pi$		$(W,1,1,1)$		$(R,1,1,1)$					
v_2	$(\perp,LF)$				$(1,LF)$				
lp_2^{-1}	$\{2\}$				$\{1\}$				
$\tau[3]^\pi$		$(W,2,1,2)$		$(W,1,1,1)$		$(R,1,1,1)$			
v_3			$(2,LF)$				$(1,LF)$		
lp_3^{-1}			$\{2\}$				$\{1\}$		
$\tau[4]^\pi$		$(W,2,1,2)$		$(W,1,1,1)$		$(R,1,1,1)$		$(R,2,1,1)$	
v_4							$(1,LO)$		$(1,NF)$
lp_4^{-1}							$\{1\}$		$\{2\}$

Table 2: Here we consider the trace $\tau = (W,1,1,1),(R,1,1,1),(W,2,1,2),(R,2,1,1)$, given in [2] as an example of a SC sequence that is *not* a trace of Lazy Caching. This example shows that τ is a trace of $\mathcal{G}_2$, i.e. τ is DSC with VW-bound 2. A DSC-reordering of τ is π, where $\tau^\pi = (W,2,1,2),(W,1,1,1),(R,1,1,1),(R,2,1,1)$. The table gives VWs $w_0,\ldots,w_4$ of $\tau[0]^\pi,\ldots,\tau[4]^\pi$ respectively that satisfy conditions 1 and 2 of definition 5 for $k=2$.

Lemma 1 *If w is a VW of σ, and $w \leq_{vw} w'$ then w' is a VW of σ.*

Proof: We show that if w is a VW of σ, then applying a single *hop* or *delete* operation gives another VW of σ; the lemma follows by induction. Let ps be such that $w \in VW(\sigma, \mathsf{ps})$.

Case: $w' = (\mathsf{v}', \mathsf{lp}') = hop(w, p, h)$. Then we have $\mathsf{lp}(p) < \mathsf{lp}'(p) = h \leq |\mathsf{v}'|$, this being the sole discrepancy between w and w'. Let $x = \min(\{i \mid lpa(\sigma, p) \leq \mathsf{ps}(i)\} \cup \{0\})$. Since $w \in VW(\sigma, \mathsf{ps})$, we have from def. 3 that $\mathsf{lp}(p) \geq x$, and thus also $\mathsf{lp}'(p) \geq x$. Thus $w' \in VW(\sigma, \mathsf{ps})$.

Case: $w' = (\mathsf{v}', \mathsf{lp}') = delete(w, h)$. Let ps' be ps with the hth entry removed. We claim that $w' \in VW(\sigma, \mathsf{ps}')$, and show that v' and lp' are compliant with the conditions of def. 3. Note that since *delete* requires $h < |\mathsf{v}|$, we have that ps' has $|\sigma|$ as the final element, and hence ps' is a legal position sequence for σ.

Let $\hat{\mathsf{v}}$ be v with $\mathsf{v}(h)$ removed, and let ps' be ps with the hth entry removed. Clearly, for all $i \in \mathbb{N}_{|\mathsf{ps}'|}$ we have $val(\hat{\mathsf{v}}(i)(b)) = \sigma(lw(\sigma, \mathsf{ps}'(i), b))$. Further, $tag_2(\hat{\mathsf{v}}(i)(b)) = O$ iff $IR(\sigma, \mathsf{ps}'(i), b)$. Thus v' and ps' are compliant with def. 3.1 and def. 3.3, since v' only differs from $\hat{\mathsf{v}}$ in tag_1 components. In general $\hat{\mathsf{v}}$ does not satisfy def. 3.2. This is because the boolean expression in def. 3.2 depends on other entries in the position sequence, which was altered. The only case where $\hat{\mathsf{v}}$ can potentially be in violation is at position h, which can be seen as follows. We have $\hat{\mathsf{v}}(h) = \mathsf{v}(h+1)$ and $\mathsf{ps}'(h) = \mathsf{ps}(h+1)$. It is possible that for some address b, we have that

$$\mathsf{ps}(h-1) < lw(\sigma, \mathsf{ps}(h), b) = lw(\sigma, \mathsf{ps}(h+1), b)$$

(taking $\mathsf{ps}(0) = 0$ if necessary) which implies both $tag_1(\hat{\mathsf{v}}(h)(b)) = N$ and $\mathsf{ps}'(h-1) < lw(\sigma, \mathsf{ps}'(h), b)$. In this case we find a noncompliance. However, *delete* returns v', which differs from $\hat{\mathsf{v}}$ by correcting precisely such scenarios. Therefore v' is compliant. Since lp' is obtained from lp by simply decrementing values where appropriate to reflect the deleted view, it follows that lp' adheres to def. 3.4. Again we conclude that $w' \in VW(\sigma, \mathsf{ps})$. ∎

Lemma 2 *Suppose $w = (\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$ and w is directly α-enabled. Then the string σ' obtained by inserting α at position $\mathsf{ps}(\mathsf{lp}(proc(\alpha)))$ in σ is serial.*

Proof: Since σ is serial, we must argue that the insertion of $\alpha = (o, p, b, v)$ maintains seriality. We case-split on $op(\alpha)$. Suppose α is a read. From def. 4 we have $val(\mathsf{v}(\mathsf{lp}(p))(b)) = v \neq \bot$, and from def. 3 we have that $val(\mathsf{v}(\mathsf{lp}(p))(b)) = val(lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b))$. Thus σ' maintains seriality in this case. Now assume α is a write. From def. 4 we have $tag_2(\mathsf{v}(\mathsf{lp}(p))(b)) = F$, hence from def. 3 we have $\neg IR(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b)$. Thus the insertion of α will not interfere with any inheritance from $lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b)$, or any other write to address b. Therefore σ' is serial in both cases. ∎

Lemma 3 *Suppose $w = (\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$ and w directly α-evolves to w'. Then w' is a VW of the string σ' obtained by inserting α at position $\mathsf{ps}(\mathsf{lp}(proc(\alpha)))$ in σ.*

Proof: Let $(o, p, b, v) = \alpha$. Let ps' be the position sequence of σ obtained by inserting $\mathsf{ps}(\mathsf{lp}(p)) + 1$ into ps at $\mathsf{lp}(p)$, and then incrementing all entries right of $\mathsf{lp}(p)$, i.e.

$$\mathsf{ps}' = \mathsf{ps}(1), \ldots, \mathsf{ps}(\mathsf{lp}(p)), \mathsf{ps}(\mathsf{lp}(p)) + 1, \ldots, \mathsf{ps}(|\mathsf{ps}|) + 1$$

Our claim is that $w' = (\mathsf{v}', \mathsf{lp}') \in VW(\sigma', \mathsf{ps}')$, and we note that $VW(\sigma', \mathsf{ps}')$ is well-defined, since from Lemma 2 we have σ' serial. We case split on o. For convenience in this proof, we introduce the function $c : \mathbb{N}_{|\mathsf{ps}'|} \to \mathbb{N}_{|\mathsf{ps}|}$ given by $c(i) = i$ when $i \leq \mathsf{lp}(p)$ or $c(i) = i - 1$ otherwise. Also, ν is the inserted view, as in def. 4.

Case $o = R$: Clearly def. 3.1 is satisfied w.r.t v', since for all addresses a we have $val(\nu(a)) = val(\mathsf{v}(\mathsf{lp}(p))(a))$. This follows from the fact that $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), a) = lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), a)$, which also implies that def. 3.2 is satisfied, since for all addresses a we have $tag_1(\nu(a)) = N$ and $\mathsf{lp}'(p) > 1$.

Now for all addresses $a \neq b$ and indexes i of ps', we have $IR(\sigma', \mathsf{ps}'(i), a) \Leftrightarrow IR(\sigma, \mathsf{ps}(c(i)), a)$. Thus, the fact that for all such a, we have $tag_2(\mathsf{v}'(i)(a)) = tag_2(\mathsf{v}(c(i))(a))$ is compliant with def. 3.3. Also, if $\ell = lr(\sigma, lw(\sigma, \mathsf{lp}(p), b)) > \mathsf{ps}(\mathsf{lp}(p))$, then we have both $IR(\sigma', \mathsf{ps}'(i), b) \Leftrightarrow IR(\sigma, \mathsf{ps}(c(i)), b)$ and $tag_2(\mathsf{v}'(i)(b)) = tag_2(\mathsf{v}(c(i))(b))$ for all indexes i of ps'. Then def. 3.3 is satisfied. However, if $\ell \leq \mathsf{ps}(\mathsf{lp}(p))$ then for each i in the nonempty set $I = \{i \mid i < \mathsf{lp}'(p) \wedge \ell \leq \mathsf{ps}'(i)\}$ we have $IR(\sigma', \mathsf{ps}'(i), b) \wedge \neg IR(\sigma, \mathsf{ps}(c(i)), b)$. In this case *unfree* has the effect that for all $i \in I$, $tag_2(\mathsf{v}'(i)(b)) = O$, which hence preserves compliance with def. 3.3.

Finally, we have def. 3.4 satisfied w.r.t ps' and lp' since $\mathsf{ps}'(\mathsf{lp}'(p)) = \mathsf{ps}(\mathsf{lp}(p)) + 1 = lpa(\sigma', p)$. A similarly simple argument handles the other processors.

Case $o = W$: Similar to the previous case, def. 3.1 is satisfied w.r.t v'. The slight complication here is that for all i in $I = \{\mathsf{lp}'(p), \ldots, j_{min}\}$ where
$j_{min} = \min(\{j \mid j > \mathsf{lp}'(p) \wedge tag_1(\mathsf{v}'(j)(b)) = L\} \cup \{|\mathsf{v}'| + 1\}) - 1$
we have $lw(\sigma', \mathsf{ps}'(i), b) = \mathsf{ps}'(\mathsf{lp}'(p)) \neq lw(\sigma, \mathsf{ps}(c(i)), b)$, however, def. 4.2 correctly sets $val(\nu(b)) = val(\sigma'(\mathsf{ps}'(\mathsf{lp}'(p)))) = v$, and *bind* sets $val(\mathsf{v}'(i)(b)) = v$ for all $i \in I$.

Now for all addresses $a \neq b$, we have $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), a) = lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), a)$, and since we have $tag_1(\nu(a)) = N$ and $\mathsf{lp}'(p) > 1$, def. 3.2 is satisfies w.r.t all such a. For address b, we have $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), b) = \mathsf{ps}'(\mathsf{lp}'(p)) = \mathsf{ps}'(\mathsf{lp}'(p) - 1) + 1$, and thus, $\mathsf{ps}'(\mathsf{lp}'(p) - 1) < lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), b)$. Thus the assignment $tag_1(val(\nu(b))) = L$ is consistent with def. 3.2.

Similar to the previous case, we have def. 3.3 satisfied. Def. 4.2 assigns $tag_2(\mathsf{v}'(i)(a)) = tag_2(\mathsf{v}(c(i))(a))$ for all addresses a and indexes i of ps', i.e. the tag_2 components are preserved. The correctness of this preservation follows from the fact that for all such a and i we have $IR(\sigma', \mathsf{ps}'(i), a) = IR(\sigma, \mathsf{ps}(c(i)), a)$.

The argument that def. 3.4 is satisfied is the same as in the previous case.

For both cases, we conclude that $w' \in VW(\sigma', \mathsf{ps}')$. ∎

Theorem 2 *Let τ have DSC-serial reordering π, w be a VW of τ^{π}, and $\alpha \in \mathcal{M}$ be such that $w \rightsquigarrow_{\alpha} w'$. Then there exists a DSC-serial reordering π' of $\tau' = \tau\alpha$ such that $\tau^{\pi'} = \tau^{\pi}$, and w' is a VW of $\tau'^{\pi'}$.*

Proof: From def. 4, there exist VWs w_1 and w_2 such that $w \leq_{vw} w_1$ and $w_2 \leq_{vw} w'$, and w_1 directly α-evolves to w_2. From Lemma 1 we have that w_1 is a VW of τ^{π}.

Choose ps to be a position sequence such that $w_1 = (\mathsf{v}_1, \mathsf{lp}_1) \in VW(\tau^{\pi}, \mathsf{ps})$, and define $(o, p, b, v) = \alpha$. Let π' be the permutation of τ' such that $\tau'^{\pi'}$ is the string obtained by inserting α into τ^{π} at position $\mathsf{lp}_1(p)$. From this definition of π' we have $\tau^{\pi'} = \tau^{\pi}$.

We must show that π' is a DSC-serial reordering of τ'. π' adheres to the processor order of τ', since π adheres to the processor order of τ, and by the def. 3 we have $\mathsf{ps}(\mathsf{lp}(p)) \geq lpa(\tau^{\pi}, p)$. Thus our construction of $\tau'^{\pi'}$ is such that π' respects $<^{po}_{\tau'}$, while Lemma 2 guarantees that $\tau'^{\pi'}$ is serial.

Finally, from Lemma 3 we have that w_2 is a VW of $\tau'^{\pi'}$, and hence by Lemma 1 we have w' is a VW of $\tau'^{\pi'}$. ∎

5. VW-BOUNDEDNESS

In this section we define the concept of the VW-bound of a DSC sequence, and show that the set of DSC protocols with a given bound k is decidable for any k.

Definition 5 (VW-bound) *Let τ be a trace such that there exists VWs $w_1, \ldots, w_\ell$ such that*

1. *for all $1 \leq i \leq \ell$ we have $w_{i-1} \rightsquigarrow_{\tau(i)} w_i$, and*
2. *there exists $k \geq 1$ such that for all $0 \leq i \leq \ell$ we have $|w_i| \leq k$,*

where w_0 is the view window of the empty trace. Then we say that τ has VW-bound *k and also we write $w_0 \rightsquigarrow_{\tau,k} w$.*

Theorem 3 *If $w_0 \rightsquigarrow_{\tau,k} w$, then τ is DSC.*

Proof: Follows from a simple induction using theorem 2 and the fact that w_0 is a VW of the empty trace. ∎

Definition 6 (DSC_k) *DSC_k is the set of all protocols $\mathcal{P}$ such that all traces of $\mathcal{P}$ have VW-bound k.*

Lemma 4 *For any integer $k \geq 1$, there is a protocol, $\mathcal{G}_k$, such that $traces(\mathcal{G}_k)$ is precisely the set of traces τ such that τ has VW-bound k.*

Proof: The state space of $\mathcal{G}_k$ is the set of all VWs w such that $|w| \leq k$; note that this is a finite set. The initial state of $\mathcal{G}_k$ is the view window w_0. The set of actions $\mathcal{A}(\mathcal{G}_k)$ of the protocol is the set $\mathcal{M}$. The set of transitions $\delta(\mathcal{G}_k)$ is $\{(w, \alpha, w') \mid w \rightsquigarrow_\alpha w'\}$.

If $\tau \in traces(\mathcal{G}_k)$, then clearly $w_0 \rightsquigarrow_{\tau,k} w$. Conversely, if τ has VW-bound k, from the definition of $\mathcal{G}_k$ and def. 5 we have that $\tau \in traces(\mathcal{G}_k)$. ∎

Theorem 4 *For any $k \geq 1$, DSC_k is decidable.*

Proof: Given a protocol $\mathcal{P}$, we can decide whether $\mathcal{P}$ in DSC_k by simple trace containment against the protocol $\mathcal{G}_k$ given in Lemma 4. ∎

6. DISCUSSION

All sequentially consistent protocols we have found in the literature are DSC with VW-bound. Furthermore, the bound is always very small. For instance, any protocol with all traces being serial is in DSC_1. The following theorem states that Lazy Caching is also VW-bounded.

Theorem 5 *Lazy Caching [2] is in $DSC_{j+P\ell+1}$, where j and ℓ are the respective capacities of the In and Out queues, and P is the number of processors.*

Proof Sketch: Here we employ some notation of [2]. The abstracted (R, i, a, d) and (W, i, a, d) events are identified with the Lazy Caching events ReadReturn$_i(d, a)$ and WriteReturn$_i(d, a)$, respectively.

For any state of Lazy Caching, we associate a finite set of possible VWs w, where $w = (\mathsf{v}, \mathsf{lp})$ is as follows. For convenience, we shift the index set of the sequence v (and hence the range of the function lp) leftwards by $j+1$, i.e. $\mathsf{v} = \mathsf{v}(-j)\ldots\mathsf{v}(0)\ldots\mathsf{v}(P\ell)$. The *val* components of $\mathsf{v}(0)$ always represents the contents of *Mem*. When $\mathsf{lp}(i) \leq 0$, this corresponds to $|In_i| = -\mathsf{lp}(i)$ and $|Out_i| = 0$. The $P\ell$ possible views to the right of $\mathsf{v}(0)$ are to accommodate the views corresponding to the $P\ell$ potential *Out* queue entries that may be buffered in the system at any given time.

Prior to performing an event (W, i, a, v), the operation $hop(w, i, h)$ must be performed for some[4] $h \geq 0$. The actual value of h is nondeterministic; the relative ordering of the views to the right of $\mathsf{v}(0)$ reflect a "prediction" made regarding the order that the associated W events are seen by *Mem* via MemoryRead actions.

Other Lazy Caching actions result in the following updates to w. ReadRequest, WriteRequest, and CacheInvalidate produce no change. MemoryWrite causes no effective change, though under our shifted view naming convention, all names would be decremented. MemoryRead inserts a new view with the same *val* components as $\mathsf{v}(0)$ immediately following $\mathsf{v}(0)$. CacheUpdate$_i$ simply increments $\mathsf{lp}(i)$.

To see that $j + P\ell + 1$ is an upper bound on $|w|$, we note that the leftmost view in v can always be deleted (without impeding any possible future events) whenever we have $\mathsf{lp}(i) > -j$ for all processors i. ∎

Thus we have a procedure to verify Lazy Caching. It is important to note, however, that bounded VWs can be employed to produce traces that Lazy Caching cannot generate. For instance, even $\mathcal{G}_2$ has traces that are not traces of Lazy Caching. Table 2 provides an example of a trace that is DSC with VW-bound 2, but is not a trace of Lazy Caching.

We note, however, that the work of this paper lies primarily in the domain of theoretical interest. As the following complexity analysis reveals, the procedure suggested by the proof of Theorem 4 is worst case exponential in the size of the description of $\mathcal{P}$ and doubly exponential in k. We first obtain an upper bound on the size of the state space of $\mathcal{G}_k$. There are $(4V)^B$ possible views, hence at most $(4V)^{Bk}$ possible view sequences in $\mathcal{G}_k$. Each may have at most k^P logical pointer functions, thus $|S(\mathcal{G}_k)| \leq k^P(4V)^{Bk} = 2^{O(k)}$. Now, to perform the check $traces(\mathcal{P}) \subseteq traces(\mathcal{G}_k)$, we must determinize $\mathcal{G}_k$ in order to complement it, producing deterministic $\mathcal{G}'_k$, which can be exponentially larger. Therefore $\left|S(\mathcal{G}'_k)\right| = 2^{2^{O(k)}}$. Letting $n = |S(\mathcal{P})|$, the usual complement-and-intersect solution to language containment may thusly yield an automaton with $n2^{2^{O(k)}}$ states to explore. Note that n itself might be exponential in the length of the description of $\mathcal{P}$; also this analysis treats P, B, and V as constants.

The objective of proving decidability of *DSC* has not quite been achieved. This result would of course follow from:

Conjecture 1 *Any protocol $\mathcal{P} \in DSC$ has VW-bound k for some k, and further k is computable given a description of $\mathcal{P}$.*

---

[4]This may seem counterintuitive, since Lazy Caching allows write events to be performed by processor i even if In_i is nonempty. However, after performing (W, i, a, v), processor i cannot perform any read event until In_i is flushed of all entries present at the time the write took place.

Here “computable” is probably far too general of a term; it is likely that k is always bounded by $|S(\mathcal{P})|$ and in practise this bound is very loose.

The intuition behind Conjecture 1 is as follows. Let $\mathcal{P}$ be a protocol; the ensuing discussion is simplified if we consider an automaton $\mathcal{P}'$ with $\mathcal{L}(\mathcal{P}') = traces(\mathcal{P})$ rather than $\mathcal{P}$ itself. For states s_1 and s_2 of $\mathcal{P}'$, let $traces(s_1)$ to be the set of traces that drive $\mathcal{P}'$ to s_1, and let $traces(s_1, s_2)$ be the set of traces that take s_1 to s_2 in $\mathcal{P}'$. Define $W = \{w \mid w \text{ is a VW of some } \tau \in traces(\mathcal{P}')\}$, and let f be a function $S(\mathcal{P}') \to 2^W$. We call f a *VW labeling function* if we have that for any $s_1, s_2 \in S(\mathcal{P}')$ we have for any $\tau_1 \in traces(s_1)$ and $\tau_2 \in traces(s_1, s_2)$ that there exists $w_1 \in f(s_1)$ and $w_2 \in f(s_2)$ such that w_1 is a VW of τ_1 and w_2 is a VW of $\tau_1\tau_2$ and $w_1 \rightsquigarrow_{\tau_2} w_2$. If $f(s)$ is finite for all states s, then f is a *finite VW labeling function*. For any DSC protocol a labeling function exists; the function $f_\infty(s) = \{w \mid w \text{ is a VW of } \tau^\pi \text{ for some } \tau \in traces(s) \text{ with DSC-reordering } \pi\}$ demonstrates this. However, f_∞ is not finite in general.

Conjecture 2 *Any protocol $\mathcal{P} \in DSC$ has a finite VW labeling function.*

Note that since any finite set of finite sets of VWs must contain a largest VW, Conjecture 2 implies Conjecture 1. (In fact, the two statements are equivalent).

Our discussion is concluded by highlighting some alternate uses of VWs. First, since VWs are generalizations of the *windows* found in [10], they could be employed in a similar fashion. This is a semi-automatic protocol verification technique in which the designer essentially augments the protocol description to periodically output a window, which summarizes the current reordering of the trace history. Verification proceeds though an automaton called the *Checker*, which observes the protocol and determines if consecutive windows are consistent; this notion of consistency is similar to our notion of VW evolution. Second, VWs can be incorporated as a mathematical tool in hand proofs. One can prove that a protocol is DSC by defining a VW labeling function for the protocol. Third, having the VW labeling function formally defined during protocol design will not only yield protocols that are “correct by construction”, but may illuminate subtle optimizations that preserve DSC while increasing protocol efficiency.

Finally, we note that (hand) proof of Lazy Caching's sequential consistency has received considerable interest from the formal methods community. Indeed, a special issue of *Distributed Computing* [6] is devoted to this problem and contains six papers, each giving a distinctive proof. A detailed proof of our theorem 5 (which implies the sequential consistency of Lazy Caching) would probably span at most two pages if the VW definitions and theory are assumed. A two page proof is arguably more succinct than any of the six in the *Distributed Computing* issue [6].

7. RELATED WORK

The protocol of $\mathcal{G}_k$ of Lemma 4 is an example of a *maximally general model*. Park and Dill have implemented maximally general models for the SPARC architecture memory models (TSO/PSO/RMO) using murφ [16, 17]. The goal of this research is to construct an *executable specification* which can be used to verify that given parallel code is correct under a memory model, as opposed to verification that a shared memory protocol implements a model.

Only two algorithmic schemes for automatic SC verification have emerged in the literature [7, 8].[5] More accurately, these verify proper subclasses of SC. Here we contrast the decided classes with *DSC*, and focus on the characterization of these classes rather than delve into the actual verification procedures.

7.1 Qadeer's Approach

Qadeer's work [8] verifies protocols of a class specified by several assumptions, and the theory is developed under the NNPC protocol model. The approach inherently supports simultaneous verification of an infinite family of protocols $\{\mathcal{P}_1, \mathcal{P}_2, \ldots\}$ where the number of data values (i.e. our parameter V) of $\mathcal{P}_i$ is i. Hence the problem solved by Qadeer is fundamentally different than ours. To make a fair comparison, then, we fix V and consider the class of protocols Q consisting of all $\mathcal{P}$ such that $\mathcal{P}$ is a protocol with V values in a family that the technique can verify.

Some of the involved assumptions are mandatory prerequisites for his technique (i.e. they are characteristics of Q), while others support symmetry reductions that deflate the complexity of the algorithm. Here we consider only the former, which are as follows:

Data independence constrains how the $\mathcal{P}_i$s relate to each other, hence this assumption has no relevance to Q, in which V is fixed.

Causality is an assumption that is primarily related to the use of *initial functions*[6]. Under our definitions (which preclude the use of initial functions) causality is an implication of SC. Thus, the causality assumption is not a significant difference between the *DSC* and Q protocol classes.

Simple witness is equivalent to the property that each trace τ has a serial reordering π that preserves the order of writes to the same address. Formally, for all indexes i and j such that $op(\tau(i)) = op(\tau(j)) = W$ and $addr(\tau(i)) = addr(\tau(j))$, we have $i < j \Leftrightarrow \pi(i) < \pi(j)$. Many SC protocols have the simple witness, however Lazy Caching is an example of one that does not.

Since Q contains NNPC protocols, Q has members that are not in DSC_k for any k, since DSC_k assumes the PC model. However Lazy Caching is a protocol that resides in DSC_k for a finite k, but is not in Q (since it lacks the simple witness). If we accept that all real SC protocols are prefix-closed and DSC, then Conjecture 1 implies that all real protocols in Q are in DSC_k for some k.

7.2 Condon and Hu's Approach

Condon and Hu's approach [7], which pertains only to PC protocols, requires that a protocol has the attributes of *tracking labels* and a *store order generator* (SOG).

Tracking labels are motivated by a property that most real protocols have: the protocol involves a set of L storage locations (where each location might be a cache or memory line, a queue or buffer entry, a field in a network packet, etc.), and protocol transitions involve copying data directly between these locations. Tracking label's allow on-the-fly inference of the inheritance relation.

[5]It was originally asserted that the Test Model Checking approach [18] could automatically verify SC. Subsequent research showed that it actually verifies a weaker class than SC [19]. As such, the approach is a valuable debugging aid, but does not provide verification of SC.

[6]An initial function *init* maps a trace τ and an address b to a data value $init(\tau, b)$. Then the notion of a *serial* trace is defined so that a trace σ may have index i such that $lw(\sigma, i, addr(\sigma(i))) = 0$ and $op(\sigma(i)) = R$, as long as $val(\sigma(i)) = init(\sigma, addr(\sigma(i)))$.

SOG is a finite state machine which must have no more states than the protocol itself. The machine takes as input a run r of the protocol, and outputs a description of the digraph defined by the *store order*. Given a serial-reordering π of the trace $\tau = trace(r)$, the store order is the partial order $<_{st}$ defined by $i <_{st} j$ iff $op(\tau(i)) = op(\tau(j)) = W$, $addr(\tau(i)) = addr(\tau(j))$, and $\pi(i) < \pi(j)$. A SOG can only exist if the protocols transitions involving actions of $\mathcal{A}(\mathcal{P}) \setminus \mathcal{M}$ are labeled in a manner that provides enough information to deduce the store order. Implicit in Condon and Hu's work is the fact that since the SOG state space is bounded in size, a procedure could theoretically enumerate all possible SOGs for a protocol, hence providing full automation. In practise, the SOG would be either elementary (for protocols with Qadeer's simple witness) or user-supplied.

Given any nontrivial protocol $\mathcal{P}$ with tracking labels and a SOG, we can derive a protocol $\mathcal{P}'$ such that $traces(\mathcal{P}) = traces(\mathcal{P}')$ and $\mathcal{P}'$ has neither tracking labels nor a SOG. Tracking labels can be foiled by introduction of an encoding mechanism when moving data between locations. For example, a protocol modeled at a level that includes use of error correction encoding of data would *not* have tracking labels. Although one could in principle identify a tracking label scheme for such a protocol, it would be difficult or perhaps impossible to concoct an algorithm that performs this identification automatically. The SOG could be disabled by renaming all actions of $\mathcal{A}(\mathcal{P}) \setminus \mathcal{M}$ with a new silent action. This modification would typically "hide" the run information needed by the SOG to construct the store order.

As an example, we again call upon Lazy Caching [2]. Lazy Caching has both tracking labels and a SOG. The SOG observes instances of the MemoryWrite action to determine the store order. We define Lazy Caching′ to be obtained from Lazy Caching via the modifications suggested above. Lazy Caching′ could perform data encoding by, say, performing a bijective mapping on values when appending to the *Out* queues, and applying the inverse mapping upon MemoryWrite (which has actually been renamed). Clearly $traces(\text{Lazy Caching}) = traces(\text{Lazy Caching}')$, yet Lazy Caching′ can be verified via Theorems 4 and 5, but not by Condon and Hu's method.

In summary, we find that our class *DSC* contains:

- all protocols in Qadeer's class *Q* that do not demonstrate "prophetic inheritance" and are prefix closed, and
- all protocols in the class of Condon and Hu, and
- natural protocols that are not in either class.

Whether the same statement can be made with respect to our class $\bigcup_k DSC_k$ is an open problem and is of course implied by conjecture 1.

8. ACKNOWLEDGEMENT

We thank Shaz Qadeer for his useful comments during the preparation of this article.

9. REFERENCES

[1] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *11th Annual IEEE Symp. on Logic in Comp. Sci. (LICS'96)*, New Brunswick, New Jersey, Jul 1996.

[2] Y. Afek, G Brown, and M. Merritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

[3] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[5] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport clocks: Verifying a directory cache-coherence protocol. In *Tenth Annual ACM Symposium for Parallel Algorithms and Architectures*, June/July 1998.

[6] Michael Merritt, editor. *Distributed Computing*, volume 12. 1999.

[7] A. Condon and A.J. Hu. Automatable Verification of Sequential Consistency. In *Thirteenth Annual ACM Symposium for Parallel Algorithms and Architectures*, pages 113–121, July 2001.

[8] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, Compaq Systems Research Center, December 2001. SRC Research Report 176.

[9] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Verifying Sequential Consistency on Shared-memory Multiprocessor Systems. In *Proceedings of the 11th International Conference on Computer-Aided Verification*, 1999.

[10] Tim Braun, Anne Condon, Alan J. Hu, Kai S. Juse, Marius Laza, Michael Leslie, and Rita Sharma. Proving Sequential Consistency by Model Checking. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, November 2001. An expanded version appeared as University of British Columbia dept. of Computer Science Tech Report TR-2001-03, http://www.cs.ubc.ca/cgi-bin/tr/2001/TR-2001-03.

[11] T. Arons. Using timestamping and history variables to verify sequential consistency. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, July 2001. volume 2102 of Lecture Notes in Computer Science.

[12] Marcelo Glusman and Shmuel Katz. Extending memory consistency of finite prefixes to infinite computations. In Kim G. Larsen and Mogens Nielsen, editors, *Proceedings of 12th International Conference on Concurrency Theory, CONCUR '01*, volume 2154 of *LNCS*, pages 411–425, Aalborg, Denmark, August 2001. Springer-Verlag.

[13] Shaz Qadeer. private communication, January 2003.

[14] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[15] Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. *SIAM J. Computing*, 26(4):1208–1244, August 1997.

[16] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Proceedings of the 1993 Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, March 1993.

[17] Seungjoon Park and David L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2):227–235, February 1999.

[18] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *Computer-Aided Verification: 10th International Conference*, pages 464–476. Springer, 1998.

Lecture Notes in Computer Science Vol. 1427.
[19] Ganesh Gopalakrishnan. A formalization of test model-checking, completeness results, and case studies. In *Workshop on Advances in Verification*, 2000. Informal workshop affiliated with the Conference on Computer-Aided Verification, 2000, with participant proceedings only. The paper is available from http://www.cs.utah.edu/formal_verification/papers/wave2000.pdf.