Automatic Verification of Sequential Consistency for Unbounded Addresses and Data Values ⋆

Jesse Bingham[1], Anne Condon[1], Alan J. Hu[1], Shaz Qadeer[2], and Zhichuan Zhang[1]

[1] Department of Computer Science, University of British Columbia
[2] Microsoft Research

Abstract. Sequential consistency is the archetypal correctness condition for the memory protocols of shared-memory multiprocessors. Typically, such protocols are parameterized by the number of processors, the number of addresses, and the number of distinguishable data values, and typically, automatic protocol verification analyzes only concrete instances of the protocol with small values (generally < 3) for the protocol parameters. This paper presents a fully automatic method for proving the sequential consistency of an entire parameterized family of protocols, with the number of processors fixed, but the number of addresses and data values being unbounded parameters. Using some practical, reasonable assumptions (data independence, processor symmetry, location symmetry, simple store ordering, some syntactic restrictions), the method automatically generates a finite-state abstract protocol from the parameterized protocol description; proving sequential consistency of the abstract model, via known methods, guarantees sequential consistency of the entire protocol family. The method is sound, but incomplete, but we argue that it is likely to apply to most real protocols. We present experimental results showing the effectiveness of our method on parameterized versions of the Piranha shared memory protocol and an extended version of a directory protocol from the University of Wisconsin Multifacet Project.

1 Introduction

Shared-memory multiprocessors are the dominant form of multiprocessing. In such systems, the processors share a single address space and interact by reading/writing to a shared memory system. A *memory model* is the correctness condition for the memory system, defining the processor-visible behavior of the system.

Sequential Consistency (SC) [20] is the archetypal memory model. Informally, SC states that every execution of the system must behave as if it were some interleaving of the individual processors' executions on a single atomic memory. For example, Fig. 1 shows an execution that is not sequentially consistent, because there is no way to interleave the executions of the two processors on a single memory and obtain the observed results. SC continues to be important both from a verification perspective as a well-defined and extensively-researched challenge problem, as well as from an implementation perspective as a memory model balancing ease-of-programming with implementation flexibility [17].

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

$$proc_1 : (\text{write } addr_2\ val_1),\ (\text{write } addr_1\ val_2),\ (\text{read } addr_2\ val_1),\ (\text{read } addr_2\ val_2)$$
$$proc_2 : (\text{write } addr_1\ val_1),\ (\text{write } addr_2\ val_2),\ (\text{read } addr_1\ val_1),\ (\text{read } addr_1\ val_2)$$

Fig. 1. Example execution that is not sequentially consistent. The values seen by the two reads on one processor imply that the other processor's second write appears to have occurred between the two reads; no interleaving can satisfy this property for both processors simultaneously. Note that the operations on each address, considered in isolation, **are** sequentially consistent, demonstrating that sequential consistency cannot be verified on a per-address basis. (In fact, the per-address operations satisfy the even stronger notion of *simple sequential consistency*, defined in Sec. 2.)

Memory systems use intricate finite-state protocols to implement the desired memory model. These protocols are notoriously complex and error-prone, because the primary objective is performance rather than simplicity. With the ascendance of finite-state model checking [9] as an automatic verification method, the verification of multiprocessor memory system protocols has been a major success story of the practical application of formal verification.

Finite-state model checking is limited, of course, to finite-state systems. In reality, memory system protocols are defined as parameterized systems — typically by the number of processors, the number of addresses, and the number of distinguishable data values. Most automatic protocol verification efforts have therefore considered only concrete instances of protocols. Furthermore, because of problems with state-space explosion, the instances verified are generally very small, e.g., for a detailed model of a typical industrial protocol, a successful model-checking run with 3 processors, 3 addresses, and 3 data values is a remarkable achievement. Far better would be a method to automatically verify an entire parameterized protocol family.

In theory, handling a parameterized number of processors is most interesting, because shared memory protocols are intended to facilitate complex interactions among processors. In practice, however, handling parameterized numbers of addresses and data values is a higher priority, because real shared-memory multiprocessors have few processors and many addresses and data values. For example, typical configurations have 2 to 8 processors, with even the largest installations having at most a few dozen processors. In contrast, even the smallest and most common configurations (e.g., a hyperthreading desktop PC) have at least 2^{32} data values and hundreds of millions of physical addresses (with much larger virtual address spaces) — far beyond the reach of the direct application of model checking for the foreseeable future.

This paper presents a fully automatic method for proving the sequential consistency of an infinite family of protocols parameterized in two dimensions: the number of addresses, and the number of data values. We consider the number of processors to be a fixed constant. Our approach relies on data independence to handle the parameterized data values; our main contribution is a means to handle parameterized addresses. Note that unlike easier-to-verify properties like seriality or linearizability [16], sequential consistency cannot be verified on a per-address basis. (See Fig. 1.) No previous fully automatable method for verifying sequential consistency can be parameterized by both the number of addresses and the number of data values. (See Sec. 7 for related work.)

Our method leverages a few common, practical assumptions about memory system protocols. Three of these — data independence, processor symmetry, and location symmetry — are standard and easily enforced syntactically. We impose some additional syntactic constraints to simplify the automatic generation of a finite-state abstract protocol from the parameterized protocol description; these are described in Sec. 4. Finally, our method verifies a slightly stronger form of SC in which writes to an address cannot be reordered. To our knowledge, all implemented SC protocols implement this stronger form. (The canonical example of a protocol that is SC, but violates this assumption is *Lazy Caching* [1].) With these restrictions, our method is, in principle, fully automatic.

2 Preliminaries

A *labelled transition system* (LTS) is a tuple $M = (S, \Sigma, I, \longrightarrow)$, where S is a set of states, Σ is a finite alphabet, $I \subseteq S$ is the set of initial states, and $\longrightarrow \subseteq S \times \Sigma \times S$ is the transition relation. The language $\mathcal{L}(M)$ of the LTS M is the subset of Σ^* defined in the usual way. Given an alphabet Σ and some string x, define $x \uparrow \Sigma$ to be the string obtained by deleting all symbols of x that are not in Σ. We extend $\uparrow$ to act on sets of strings in the obvious way.

Denote by $\mathbb{N}$ the set of positive integers, and by $\mathbb{N}_n$ the set $\{1,\ldots,n\}$. For sets P, A, and V, let $\mathit{MemEvents}(P,A,V)$ be the set $\{R,W\} \times P \times A \times V$. Then $\mathit{MemEvents}(\mathbb{N},\mathbb{N},\mathbb{N})$ is called the set of *memory events*, also denoted simply *MemEvents*. An occurrence of memory event (R, p, j, d) is meant to represent processor p reading value d from address j, we call such an event a *read*. Similarly the *write* event (W, p, j, d) indicates processor p writing value d to address j. We call a finite string over *MemEvents* a *trace*. A shared memory protocol, hereafter simply *protocol*, is formalized as a LTS $\mathcal{P} = (S, \mathit{MemEvents}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_v) \cup E, I, \longrightarrow)$ for some $n, m, v \geq 1$, where S is finite and E (the *silent* action labels) is disjoint from *MemEvents*. The quantities n, m, and v are respectively denoted $\mathit{Procs}(\mathcal{P})$, $\mathit{Addrs}(\mathcal{P})$, and $\mathit{Vals}(\mathcal{P})$. Intuitively, these quantities are respectively the number of processors, number of memory addresses, and number of data values (per address) processed by the protocol. We define *PROTS* to be the set of all protocols. For a protocol $\mathcal{P}$, define $\mathit{traces}(\mathcal{P})$ to be $\mathcal{L}(\mathcal{P}) \uparrow \mathit{MemEvents}$.

We aim to verify correctness of an infinite set of related protocols called a protocol *family*, defined as follows. For fixed $n \geq 1$, an *n-processor family* is a function $\mathcal{F} : \mathbb{N} \times \mathbb{N} \to \mathit{PROTS}$ where for all $m, v \geq 1$, $\mathit{Procs}(\mathcal{F}(m,v)) = n$, $\mathit{Addrs}(\mathcal{F}(m,v)) = m$, and $\mathit{Vals}(\mathcal{F}(m,v)) = v$.

A trace is said to be *serial* if every read event has the same value as the last write to the same address, and such a write always exists.[3] A trace σ is said to be *sequentially consistent* (SC) iff there exists a trace σ' (of the same length) such that (1) $\sigma' \uparrow \mathit{MemEvents}(\{p\},\mathbb{N},\mathbb{N}) = \sigma \uparrow \mathit{MemEvents}(\{p\},\mathbb{N},\mathbb{N})$ for each $p \geq 1$, and (2) σ' is serial. We call such a σ' a *serial reordering* of σ. Furthermore, σ is said to be *simple SC* (SSC) if there exists σ' with the above two properties plus the additional property (3) $\sigma' \uparrow (\{W\} \times \mathbb{N} \times \{j\} \times \mathbb{N}) = \sigma \uparrow (\{W\} \times \mathbb{N} \times \{j\} \times \mathbb{N})$ for each $j \geq 1$; here σ' is called a *simple serial reordering*. Intuitively, SC says that there must exist a reordering that is serial and preserves the per-processor order. SSC adds the requirement that the ordering of writes to each address is also preserved in the reordering. A protocol is said

[3] The *last write to the same address* is the rightmost write that is left of the read in question and has the same address, if the trace were written out from left to right as usual.

to be serial, SSC, or SC, if all of its traces are serial, SSC, or SC respectively. Similarly, an n-processor family is said to have any of these properties if all of its constituent protocols have the respective property.

3 The Big Picture

Our aspiration is to algorithmically verify that an n-processor family $\mathcal{F}$ is SSC. This section presents Theorem 1, which allows us to soundly reduce the proof that $\mathcal{F}$ is SSC to checking SSC of a protocol Q, where $Addrs(Q) = Procs(Q) = n$, and $Vals(Q) = 3$, provided that an infinite number of projected trace containments hold between certain members of $\mathcal{F}$ and Q (see condition 2 of Theorem 1). However, in Sec. 5 we show that if $\mathcal{F}$ is expressed in a certain formalism, then we can effectively produce a Q for which these containments hold "by construction". SSC of Q can be checked algorithmically using known methods based on model-checking [10, 29, 8, 7].

In Sec. 3.1 we define three assumptions that are required by Theorem 1; the theorem itself is presented in Sec. 3.2.

3.1 Assumptions

Here we define three common protocol assumptions: location symmetry (LS), processor symmetry (PS), and data independence (DI).

For a permutation λ on $\mathbb{N}$ define λ^{proc} to be the function on $MemEvents(n,m,v)$ specified by $\lambda^{proc}((op,p,j,d)) = (op,\lambda(p),j,d)$. Similarly, define $\lambda^{addr}((op,p,j,d)) = (op,p,\lambda(j),d)$. We extend λ^{proc} and λ^{addr} to have domain and range $MemEvents^*$ in the obvious way. A protocol $\mathcal{P}$ is *location symmetric (LS)* if for every permutation $\lambda : \mathbb{N}_{Addrs(\mathcal{P})} \to \mathbb{N}_{Addrs(\mathcal{P})}$ we have $\sigma \in traces(\mathcal{P})$ implies $\lambda^{addr}(\sigma) \in traces(\mathcal{P})$. Similarly, $\mathcal{P}$ is *processor symmetric (PS)* if for every permutation $\lambda : \mathbb{N}_{Procs(\mathcal{P})} \to \mathbb{N}_{Procs(\mathcal{P})}$ we have $\sigma \in traces(\mathcal{P})$ implies $\lambda^{proc}(\sigma) \in traces(\mathcal{P})$. A family is said to be location symmetric or processor symmetric if all protocols in its image have the respective property.

Intuitively, *data independence* (DI) in a system means that variables of a certain type can only be nondeterministically assigned, copied, and outputted [32]. When the system is a protocol, and the type is data values, one can define how DI manifests at the trace level. Qadeer [29] gives the following trace level definition of DI. A trace is called *unambiguous* if it has the feature that no two writes to the same address write the same value. Given $m,v,v' \geq 1$, we call a function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \to \mathbb{N}_v$ a *renaming function*, and define $\lambda^{val}((op,p,j,d)) = (op,p,j,\lambda(j,d))$. We extend λ^{val} to traces in the obvious way. Then an n-processor family $\mathcal{F}$ is said to be DI if for all $m,v \geq 1$ and traces σ, we have $\sigma \in traces(\mathcal{F}(m,v))$ if and only if there is $v' \geq 1$, an unambiguous trace $\sigma' \in traces(\mathcal{F}(m,v'))$, and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \to \mathbb{N}_v$ such that $\sigma = \lambda^{val}(\sigma')$.

3.2 Reduction to Finite-State SSC

Theorem 1. *Let $\mathcal{F}$ be an n-processor family that is processor symmetric, location symmetric, and data independent. If there exists a protocol Q such that*

1. *Q is SSC, and*

2. *For all $m > n$ we have that traces$(\mathcal{F}(m,3)) \upharpoonright MemEvents(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3) \subseteq traces(Q)$*

then $\mathcal{F}(m,v)$ is SSC for all $v \geq 1$ and $m > n$.

Proof: The detailed proof is available from:

`http://www.cs.ubc.ca/~jbingham/bchqz04-proofs.pdf`

The proof relies on machinery developed in [29] to show that, under the LS, PS, and DI assumptions, if there exists $\sigma \in traces(\mathcal{F}(m,v))$ such that σ is not SSC, we can detect that σ is not SSC by considering only $\sigma \upharpoonright MemEvents(\mathbb{N}_n, A, \mathbb{N}_v)$, where $A \subset \mathbb{N}$ has cardinality n. By exploiting the symmetry assumptions, we can ensure that such a σ exists for which $A = \mathbb{N}_n$. Furthermore, we can ensure such a σ exists using only 3 data values. Hence, if Q is SSC, and the projected traces of all family members are contained in $traces(Q)$, then all family members must be SSC. □

We note that Theorem 1 does not allow concluding that $\mathcal{F}(m,v)$ is SSC for $1 \leq m \leq n$. This is not a deficiency, since we may verify correctness of these members via a finite number of model checks. Practically, these are the uninteresting cases, since multiprocessors always have many more addresses than processors.

Suppose we are given $\mathcal{F}$ and wish to determine if a candidate protocol Q exists for which the two conditions of Theorem 1 hold. Our approach involves automatically constructing a candidate Q such that condition 2 is guaranteed to hold; this construction involves abstracting $\mathcal{F}$ in some sense. The automatic construction of Q is possible because of the formalism we use to describe $\mathcal{F}$, presented in Sec. 4. Condition 1 is then checked algorithmically using known methods. If this check is successful, the conclusion of Theorem 1 follows. Otherwise, the approach has failed, and we can draw no conclusions; in other words the approach is *sound* but *incomplete*. Hence, to argue for the applicability of this approach, one must argue that real protocol families

1. adhere to the LS, PS, and DI assumptions, and
2. can be expressed in our formalism of Sec. 4, and
3. won't yield false negatives, i.e. if the family is SSC then the approach succeeds.

It is widely accepted that real protocols satisfy property 1 [29]. Our formalism of Sec. 4 is quite general, encompassing all real protocols that we have encountered. For instance, all of the protocols [1, 5, 22, 6, 19, 14] are expressible in our formalism. In support of item 3, we present successful experiments on two challenging protocols in Sec. 6.

4 A Protocol Description Formalism

To describe an automatic construction of a candidate finite-state protocol Q from a parameterized protocol description, we must choose some sort of protocol description formalism. Here, we will assume that the n-processor family $\mathcal{F}$ is expressible in a very general syntax based on first order logic that is inspired by the *bounded-data parameterized systems* of [27]. We have tuned our formalism to provide enough expressiveness for the real protocol descriptions we have encountered, while still allowing the efficient and automatic generation of a sufficiently finely abstracted protocol Q. For the sake of perspicuity, we will treat the number of data values as being fixed at 3, since condition

2 of Theorem 1 considers such family members. Standard restrictions can be imposed on our formalism to ensure that the family is PS, LS, and DI. For instance, DI can easily be enforced by syntactic constraints as observed by Wolper and others [32, 25, 29], and symmetric types such as Murφ scalarsets [18] can be used to ensure PS. LS is inherent to the syntax [27].

4.1 Syntax

We assume three sets of variables $X = \{x_1, \ldots, x_{|X|}\}$, $Y = \{y_1, \ldots, y_{|Y|}\}$, and $Z = \{z_1, \ldots, z_{|Z|}\}$. For a set D, let $Z[D]$ denote the variable set $\{z_i[d] \mid 1 \leq i \leq |Z| \wedge d \in D\}$ and let $\mathsf{Vars}(D) = X \cup Y \cup Z[D]$. Priming any of these sets has the effect of priming all constituent variables; semantically, primed variables will represent the next state. The variables of X are Booleans, while the variables in Z are arrays of Booleans indexed by addresses. The variables of Y will range over addresses, hence we call these variables *address ranged variables* (ARVs). Of course non-Boolean finite types and arrays of such can be encoded in this framework. In shared memory protocols, typical (though by no means exhaustive) examples of the variables in X, Y, and Z, are respectively fields corresponding to processor IDs or message types in messages, fields storing addresses in messages, and the permission bits and data value associated with each address in a local cache or main memory.

We will employ auxiliary ARVs a and a_1 to quantify over, and let ARVars denote the set $Y \cup Y' \cup \{a, a_1\}$. Define *quantifier-free actions* (QFA) as formulas with syntax:

$$\Phi \ ::= \ x \mid z[a] \mid z[a_1] \mid \langle \alpha = \beta \rangle \mid \neg\Phi \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi)$$

where $x \in X \cup X'$, $z \in Z \cup Z'$ and $\alpha, \beta \in \mathsf{ARVars}$. For QFA ϕ, we write $\phi(a)$ (resp. $\phi(a, a_1)$) to emphasize that the set of auxiliary ARVs appearing in ϕ is a subset of $\{a\}$ (resp. subset of $\{a, a_1\}$). We call upon a set of action labels *Labels*, of which we require $(\{R, W\} \times \mathbb{N}_n \times \mathbb{N}_3) \subseteq \mathit{Labels}$. The transition relation of $\mathcal{F}$ must be expressible as a set:

$$\{r_\ell(a) \mid \ell \in \mathit{Labels}\}$$

where, for each $\ell \in \mathit{Labels}$, r_ℓ is a formula of the form

$$r_\ell(a) = \phi_\ell(a) \wedge \forall a_1 : \psi_\ell(a, a_1). \tag{1}$$

Here, $\phi_\ell(a)$ and $\psi_\ell(a, a_1)$ are arbitrary QFAs and we call formulas of the form (1) *restricted actions*. The initial state predicate *Init* must be of the form:

$$\mathit{Init} = \forall a : \mathit{init}(a)$$

for some QFA $\mathit{init}(a)$ that does not contain any primed variables.

Intuitively, a transition in a protocol expressed in our formalism must satisfy $r_\ell(a)$ for some ℓ and some "distinguished" address a. $\phi_\ell(a)$ dictates what happens to state related to address a, i.e. ath entries of arrays, while the conjunct $\forall a_1 : \psi_\ell(a, a_1)$ dictates the uniform effect on all other addresses. This restriction accords exactly with what we have observed in real protocol descriptions. We find that real protocol transitions have the property that ARVs are referenced and/or modified at no more than a single index a, with the exception that some more complex transitions will also modify entries at all other indices in some homogeneous way, hence the inclusion of $\forall a_1 : \psi_\ell(a,a_1)$ in restricted actions. Usually, $\psi_\ell(a,a_1)$ will simply state that if $\neg\langle a = a_1\rangle$, then the a_1th entries of arrays are left fixed. However, several transitions in one of the protocols we experimented with have a more involved ψ_ℓ, e.g., a state change for one address forces all other addresses to abandon an optimization mode. A theoretical limitation of our formalism is that the auxiliary ARVs a and a_1 are the only ARVs that can be used to index into arrays. However, typical instances of indexing using another ARV $y \in Y$ can be performed by, for example, $\langle y = a\rangle \wedge \mathit{some\_array}'[a]$.

4.2 Semantics

In this section we formally define the set of LTSs represented by our family syntax.

Let s be a valuation of the variables $\mathsf{Vars}(D)$ such that variables of X and $Z[D]$ are assigned Boolean values, and variables of Y are assigned values of some type R, and let s' be a valuation to $\mathsf{Vars}(D)'$ with analogous typing. Then we call s a D,R-*valuation* and (s,s') a D,R-*valuation pair*, respectively. Intuitively, D is the index set for the arrays in Z, and R is the type of the variables in Y; for the protocols of $\mathcal{F}$ these will be the same, but in Sec. 5 we construct a protocol for which they differ. For D,R-valuation pair (s,s') and restricted action $\theta(a)$, we write $(s,s') \vDash_m \theta(j)$ if $\theta[j/a]$ is satisfied when variables are valuated by (s,s'), and quantified ARVs are taken to range over $\mathbb{N}_m$. If θ is a QFA, we may omit the subscript on $\vDash$.

For each $m \geq 1$, our syntax defines a LTS $\mathcal{F}(m,3) = (S,L,I,\longrightarrow)$ where

- S is the set of all $\mathbb{N}_m,\mathbb{N}_m$-valuations
- $L = \mathit{Labels} \times \mathbb{N}_m$. In a slight abuse, we will identify the memory event $(op,p,j,v) \in \mathit{MemEvents}$ with $((op,p,v),j) \in L$.
- $I = \{s \mid s \vDash_m \mathit{Init}\}$
- $\longrightarrow$ is the set of all tuples $(s,(\ell,j),s')$ such that $(s,s') \vDash_m r_\ell(j)$ and $j \in \mathbb{N}_m$.

5 A Candidate Q

Here we define a candidate Q for Theorem 1, which can be viewed as a modified version of $\mathcal{F}(n,3)$; hereafter Q will refer to such. These modifications involve syntactic transformations; the transition relation and initial state assertion can easily be realized automatically given $\{r_\ell \mid \ell \in \mathit{Labels}\}$ and Init. Intuitively, the modified protocol is a finite-state abstraction of the protocols in $\mathcal{F}(n,3)$, where everything related to addresses greater than n has been conservatively abstracted away. Note, however, that our abstraction is a finer abstraction than the typical abstract interpretation [11] in which addresses greater than n are replaced by an information-destroying $\top$ value that propagates throughout the interpretation. We need our more accurate abstraction to successfully verify real protocols.

In Q, address-ranged variables have type $\mathbb{N}_n \cup \{\xi\}$. The ξ symbol represents addresses in $\mathbb{N} \setminus \mathbb{N}_n$. Counter-intuitively, the arrays in Z will still be indexed by $\mathbb{N}_n$ because the variables of $Z[\{\xi\}]$ are existentially quantified out; for brevity we will let

$\vec{z}[\xi]$ denote the variable list $z_1[\xi], \ldots, z_{|Z|}[\xi], z'_1[\xi], \ldots, z'_{|Z|}[\xi]$ throughout the paper. An important part of the transformation is the operator $sub(\cdot)$. For any action r, the action $sub(r)$ is obtained by performing the following substitution: each occurrence of $\langle \alpha = \beta \rangle$ falling under an odd number of negations (where $\alpha, \beta \in \mathsf{ARVars}$) is replaced with[4]

$$(\langle \alpha = \beta \rangle \wedge \neg(\langle \alpha = \xi \rangle \wedge \langle \beta = \xi \rangle))$$

For restricted action $\theta(a)$ and integer $j \in \mathbb{N}_n \cup \{\xi\}$, we write $(s, s') \vDash_Q \theta(j)$ if $\theta[j/a]$ is satisfied when variables are assigned by (s, s'), and the universal quantifier in θ is taken to range over $\mathbb{N}_n \cup \{\xi\}$.

We now define $Q = (S_Q, L_Q, I_Q, \longrightarrow_Q)$:

- S_Q is the set of all $\mathbb{N}_n, (\mathbb{N}_n \cup \{\xi\})$-valuations.
- $L_Q = \mathit{Labels} \times (\mathbb{N}_n \cup \{\xi\})$. We note that labels of the form $((o, p, v), \xi)$ are not in *MemEvents*.
- $I_Q = \left\{ s \mid s \vDash_Q \exists\, \vec{z}[\xi] : sub(\mathit{Init}) \right\}$. The existential quantification notation $\exists z : f$ where z is a Boolean variable and f is a formula is simply syntactic sugar for $f[\mathsf{tt}/z] \vee f[\mathsf{ff}/z]$.
- $\longrightarrow_Q$ is the set of all triples $(s, (\ell, j), s')$ such that $j \in \mathbb{N}_n \cup \{\xi\}$ and

$$(s, s') \vDash_Q \exists\, \vec{z}[\xi] : sub(r_\ell(j)) \tag{2}$$

The existential Boolean quantification in (2) blows up the formula by a factor of $2^{2|Z|}$ in the worst case. However, in practice we find that only a small number of the variables $\vec{z}[\xi]$ are actually mentioned in r_ℓ, which mitigates this effect. For example, the only dependencies on $\vec{z}[\xi]$ might be the 2 or 3 bits representing the access permissions at the ξth entry of a single processor's cache.

We conclude this section by asserting that the trace set of Q overapproximates that of $\mathcal{F}(m, 3)$ for all $m > n$, i.e. Q satisfies condition 2 of Theorem 1.

Theorem 2. *$traces(\mathcal{F}(m,3)) \upharpoonright MemEvents(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3) \subseteq traces(Q)$ for all $m > n$.*

Proof: The detailed proof is available from:

`http://www.cs.ubc.ca/~jbingham/bchqz04-proofs.pdf`

The construction of Q naturally corresponds to an abstraction function mapping concrete states from $\mathcal{F}(m, 3)$ to abstract states in Q. This mapping turns out to be a weak simulation relation, and the projected trace containment follows. □

6 Experimental Results

To evaluate our technique, we experimented with two protocols, which we call PIR and DIR. Both of these protocols are SSC but not serial, hence trace reordering requirements

[4] This syntactic substitution gives the same effect on the transition relation as the usual abstract interpretation with ξ conservatively abstracting the values greater than n. (The existential quantification of the Z variables is the key difference between our abstraction and the typical one.)

are nontrivial. PIR is a simplified abstraction of the Piranha protocol [5]. Our implementation is consistent with the details of the simplification presented in [29]. DIR is a simplification of a directory based protocol with Scheurich's optimization [30]. Our implementation is based on, but is simpler than, the description of Braun et al. [8], which was obtained from the University of Wisconsin Multifacet group. We explain part of the design of both protocols here and describe the ways in which our implementation of DIR is simpler than the description of Braun et al. [8].

In both PIR and DIR, each processor p has a cache that contains, for certain memory locations j, a data value $\mathrm{d}(p,j)$ and an access permission $\mathrm{access}(p,j)$. The permission may be modifiable (*M*), shared (*S*), or invalid (*I*). Processor p may read the data value for location j if $\mathrm{access}(p,j)$ is S or M, and may write (change) the data value if $\mathrm{access}(p,j)$ is M.

In PIR, in addition to the caches, the system state has a queue per processor and an owner per memory location. The state maintains the invariant that the owner of location j is either some processor p for which $\mathrm{access}(p,j) = S$ or M, or the owner is null. Requests of a processor to change its access level or to get a data value for a location are not modeled explicitly. Rather, if $\mathrm{owner}(j) = p$, in one transition (step) of the protocol the triple $(\mathrm{d}(p,j),j,X)$ may be placed on the queue of any processor $p' \neq p$, in which case an invalidate message (*INV*, j) is placed on the queues of any processor p'' (other than p and p') with $\mathrm{access}(p'',j) \neq I$. Also, $\mathrm{owner}(j)$ is set to null and $\mathrm{access}(p,j)$ is set to INV. In a later transition, when triple (d,j,X) is at the head of p''s queue, the triple may be removed from the queue, in which case $\mathrm{d}(p',j)$ is set to d, $\mathrm{access}(p',j)$ is set to M, and $\mathrm{owner}(j)$ is set to p'. Also, if (*INV*,j) is at the head of the queue of p'', in one transition the message may be removed from the head of the queue, in which case $\mathrm{access}(p'',j)$ is set to I. Other access permission changes are modeled similarly.

The state space of PIR is relatively small because requests of processors are not modeled explicitly, and moreover, no directory is used to store the set of processors with shared access to a memory location. However, the use of queues ensures that the traces generated by PIR are interesting and realistic, in the sense that if in some trace, the lth event σ_l is a read to location j which inherits its value from an earlier write event σ_k, then arbitrarily many operations (indeed, arbitrarily many write operations to location j) may separate σ_k from σ_l in the trace.

The DIR protocol contains many low level details that are not modeled by PIR. In DIR, a directory maintains information on which processors have exclusive or shared access to a memory location. Each processor and the directory has both a request *and* a response input queue, with different uses. For example, invalidate messages to processors are placed on the request input queue, whereas messages containing data values are placed on the response input queue.

In order to get exclusive access to a memory location, processor p sends a request to the directory, which arranges for the data value and the number of current sharers of the location to be sent to p. Before reading the data value, p must wait to receive acknowledgement messages from each current sharer. Because of the multiple queues, several race conditions can arise. For example, the directory might authorize processor p' to have exclusive access to location j after p, and p may receive a request to send j's data value to p' before p has even received the data value itself. As a result, a processor has 9 permission levels per location in its cache in addition to the I,S, and M levels. For example, $\text{access}(p,j)$ may be *IMS*, indicating that p requested M access when $\text{access}(p,j)$ was I, p has not yet transitioned to M access, but already another request for p to downgrade to S access has been received.

To keep the state space within manageable proportions, our implementation of the DIR protocol does not model a certain queue of Braun et al. [8]. This queue is used for transmission of data within a processor, rather than between processors, and its removal does not significantly change the protocol.

Both PIR and DIR were modeled at a fairly detailed level, resulting in substantial descriptions in the Murφ language (165 and 1397 lines of code, respectively, excluding comments). The Murφ language is more expressive than the formalism of Sec. 4, but the protocol family implementations used only constructs that conform to this formalism. Although the construction in Sec. 4 is obviously automatable, we do not yet have an implementation for the Murφ language, so we performed the construction by hand, exactly following the described syntactic transformations.

In addition, for the DIR protocol, the verification runs against the "automatically"-generated Q spaced out, so the protocol was manually abstracted further, yielding Q'; the numbers in Table 1 refer to the completed verification runs against Q'. The essential difference between Q and Q' is that in the latter, all fields (except for the *address* field) in messages pertaining to the abstracted address ξ were abstracted. The same approach was taken with a local record at each processor called the *transaction buffer*. Since Q' is an over-approximation of Q, the fact that the verification succeeded for Q' proves that verification would have succeeded for Q, *had we had sufficient memory resources*. In other words, our automatic construction of Q was accurate enough to prove SSC of the protocol family. In general, one could envision a tool that automatically attempts such additional abstractions if model checking the original Q is intractable.

Table 1 gives our experimental results. The 2-processor runs were successful, showing that our generated abstraction is accurate enough, even for these protocols with non-trivial reordering properties (which are therefore hard-to-prove sequentially consistent).

7 Related Work

There is a rich and successful literature pertaining to the verification of assorted safety and liveness properties on non-parameterized protocols, which we do not have space to summarize. We focus here on work pertaining to verification of SC for *parameterized families* of protocols. The problem of determining whether a finite state protocol is SC is undecidable [2], and so clearly also is the problem of determining whether a family of parameterized protocols is SC. Therefore, all methods are necessarily incomplete.

Some works verify weaker properties than sequential consistency, over parameterized memory protocols. For example, McMillan [23] uses compositional model checking to generate invariants that can be used to verify safety and liveness properties of the FLASH cache coherence protocol. Pong and Dubois [28], Delzanno [12], and Emerson and Kahlon [13] have produced automatic methods to verify safety properties over parameterized memory protocols, but these methods model the protocols at a very high level of abstraction, where implementation details are hidden. Pnueli et al. [27] and

Protocol	$k=1$				$k=2$			
	#states	time	depth	prob	#states	time	depth	prob
PIR ($q=1$)	49365	7	18	≈ 0	138621	9	20	≈ 0
PIR ($q=2$)	3782880	100	21	≈ 0	10558306	278	23	≈ 0
PIR3($q=1$)	125865495	9244	36	≤ 0.000024	374557312	25640	38	≤ 0.021101
DIR ($q=1$)	171088424	49660	53	≤ 0.000013	375967684	110211	59	≤ 0.000324

Table 1. Results for the PIR (Piranha) and DIR (Wisconsin Directory) Protocols. All runs are with the number of processors $n=2$, except for PIR3 with $n=3$. We experimented with two versions of PIR: with processors' queue depth $q=1$ and $q=2$. We use a method of Qadeer [29] to prove SSC of our generated finite-state abstract protocol. This method requires separate model-checking runs for each $k=1,\ldots,n$, to check for cycles of length $2k$ in a graph of ordering constraints. Times are in seconds, on a 2Ghz Intel Xeon with 4GB memory running Linux. "prob" is an upper-bound on the probability of missing states due to hash compaction, as reported by Murφ [31] (40 bit hashes for PIR; 42 bits for DIR). The $n=2$ runs concluded successfully, enabling us to conclude SSC for each example over all address counts m and data value counts v where $m > n = 2$. The PIR3 results are a preliminary attempt with $n=3$ processors. For PIR3 with $k=2$, we used 35-bit hashes. We have not yet completed a $k=3$ run at press time.

Arons et al. [4] present a general, sound-but-incomplete method for verifying parameterized systems. This work is similar in spirit to ours (and inspired our specification formalism) in that an incomplete procedure is used to guess a candidate, whose verification proves the property for the entire family. Their method, however, attempts to derive an inductive invariant for proving simple assertions, whereas ours derives an abstract protocol that preserves violations of SSC. All of these works can handle protocol families with parameterized number of processors, which we cannot. On the other hand, our method proves SSC rather than weaker safety properties, and applies to any protocol that can be expressed in the syntax of Sec. 4.

Several proofs of SC for parameterized protocols are manual in nature. Some [21, 26, 3] use theorem-provers, while others use formal frameworks to provide a rigorous proof [24]. In either case, a human must develop insights needed to build up the proof, and the process is quite time-consuming.

Qadeer's method [29] for verifying sequential consistency of finite-state cache coherence protocols actually provides a parameterized proof in one dimension, namely data values. Our work can be viewed as an extension of Qadeer's to two dimensions (data values and addresses). The only other approach known to us that uses model checking to verify sequential consistency of parameterized families of protocols is that of Henzinger et al. [15]. Their semi-automatic method, which builds on a method for structural induction on processes, can handle families of protocols that are parameterized in the number of processors, in addition to the number of addresses and data values. However, a limitation of their method is that its soundness relies on the assumption that the protocol to be verified is *location monotonic*, in the sense that any trace of the system projected onto a subset of locations is a trace of the system with just that subset of locations. Henzinger et al. do not provide a method (automated or otherwise) for testing location monotonicity of a parameterized family of protocols. (Nalumasu [25] does provide a framework for expressing protocols that guarantees location monotonicity, but the framework has very limited expressiveness.) Moreover, the Henzinger et al. method works only for protocols whose set of traces can be reordered by a finite state machine to form serial traces, a restriction that typically holds in practice only for protocols that are already fully serial. While protocols with traces that are not in simple SC can be handled, many protocols do not have finite-state observers; examples include both protocols described in this paper and the lazy caching protocol of Afek et al. [1]. (Although Henzinger et al. state in their paper that the lazy caching protocol has a finite state observer, this is only true of their simplified version.) In summary, while the Henzinger et al. method does allow the number of processors to be parameterized (unlike ours), their method is not automatic and applies to a limited protocol class that excludes important real protocols.

References

1. Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Prog. Lang. and Sys.*, 15(1):182–205, 1993.
2. R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *11th IEEE Symp. on Logic in Comp. Sci.*, pages 219–229, 1996.
3. T. Arons. Using timestamping and history variables to verify sequential consistency. In *Computer-Aided Verification: 13th Int'l. Conf.*, pages 423–435. Springer, 2001. LNCS Vol. 2102.
4. T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Computer-Aided Verification: 13th Int'l. Conf.*, pages 221–234. Springer, 2001. LNCS Vol. 2102.
5. L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *27th Int'l. Symp. on Comp. Arch.*, pages 282–293, 2000.
6. E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *26th Int'l. Symp. on Comp. Arch.*, pages 294–304, 1999.
7. J. D. Bingham, A. Condon, and A. J. Hu. Toward a decidable notion of sequential consistency. In *15th ACM Symp. on Parallel Algorithms and Architectures (SPAA03)*, pages 304–313, 2003.
8. T. Braun, A. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. In *IEEE Int'l. High Level Design Validation and Test Workshop (HLDVT)*, pages 103–108, 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.
9. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, Springer, 1981. LNCS Vol. 131.
10. A. Condon and A. J. Hu. Automatable verification of sequential consistency. In *13th ACM Symp. on Parallel Algorithms and Architectures (SPAA01)*, pages 113–121, 2001.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symp. on Princ. of Prog. Lang.*, pages 238–252, 1977.
12. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer-Aided Verification: 12th Int'l. Conf.*, pages 53–68. Springer, 2000. LNCS Vol. 1855.

13. E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *12th Conf on Correct Hardware Design and Verification Methods (CHARME)*, pages 247–262, 2003.
14. G. Gopalakrishnan, R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu. Formal modeling and validation applied to a commercial coherent bus: a case study. In *Conf on Correct Hardware Design and Verification Methods (CHARME)*, pages 48–62, 1997.
15. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *Computer-Aided Verification: 11th Int'l. Conf.*, pages 301–315. Springer, 1999. LNCS Vol. 1633.
16. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys.*, 12(3):463–492, 1990.
17. M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.
18. C. N. Ip and D. L. Dill. Better verification through symmetry. In *Int'l. Conf. on Computer Hardware Description Languages*, pages 87–100, 1993.
19. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *21st Int'l. Symp. on Comp. Arch.*, pages 302–313, 1994.
20. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
21. P. Loewenstein and D. L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. *Formal Methods in System Design*, 1(4):355–383, 1992.
22. M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: low-latency coherence on unordered interconnects. In *Int'l Symp. on Comp. Arch.*, pages 182–193, 2003.
23. K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Conf. on Correct Hardware Design and Verification Methods (CHARME)*, pages 179–195. Springer, 2001. LNCS Vol. 2144.
24. M. Merritt, ed. *Distributed Computing*, 12(2-3), 1999. Special issue devoted to proving sequential consistency of lazy caching.
25. R. Nalumasu. *Formal Design and Verification Methods for Shared Memory Systems*. PhD thesis, University of Utah, 1999.
26. S. Park and D. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th Symp. on Parallel Algorithms and Architectures*, pages 288–296, 1996.
27. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th Int'l. Conf., (TACAS)*, pages 82–97. Springer, 2001. LNCS Vol. 2031.
28. F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.
29. S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Trans. on Parallel and Distributed Systems*, 14(8):730–741, August 2003. Also appeared as Compaq Systems Research Center Report 176, December 2001.
30. C. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989. USC Tech Report CENG 89-19.
31. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *Conf. on Correct Hardware Design and Verification Methods (CHARME)*, pages 206–224, 1995.
32. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *13th Symp. on Princ. of Prog. Lang.*, pages 184–192, 1986.