Heuristics for Complexity-Effective Verification of a Cache Coherence Protocol Implementation

Dennis Abts*
dabts@cray.com

Ying Chen†
wildfire@ece.umn.edu

David J. Lilja†
lilja@ece.umn.edu

*Cray Inc.
P.O. Box 5000
Chippewa Falls, Wisconsin 54729

†University of Minnesota
Electrical and Computer Engineering
Minnesota Supercomputing Institute
Minneapolis, Minnesota 55455

Abstract

Verifying the correctness of a shared-memory multiprocessor cache coherence protocol, and its implementation in silicon, is an extraordinarily complex and time-consuming task. The detailed formal verification model developed for the Cray X1 cache coherence protocol, for instance, produces a search space with over 214 million reachable states. Exhaustively searching this space for errors in the protocol and its implementation is computationally prohibitive. To address this problem, we describe a novel method for extracting traces called "witness strings" from the formal verification process. These witness strings then are executed on a logic simulation of the hardware that implements the protocol. To reduce the number of states that need to be explored to discover an error in a coherence protocol using this verification environment, we introduce and compare several search heuristics. We show that the min-max-predict *search heuristic consistently outperforms both breadth-first search (BFS) and depth-first search (DFS) for 12 protocol errors manually embedded in the Cray X1 and Stanford DASH coherence protocols. In many cases, this heuristic was able to reduce the number of states that needed to be searched by a factor of 50-100.*

1 Introduction

Shared memory multiprocessors must enforce mutual exclusive access over the shared main memory in order to present a consistent view of the memory hierarchy to the programmer. The memory consistency model (MCM) described by the instruction set architecture (ISA) provides a set of rules that the programmer (and compiler) must follow to ensure that a parallel program executes correctly. The cache coherence protocol is a fundamental ingredient of the MCM responsible for propagating writes. Showing that a cache coherence protocol is correct is nontrivial, as there are many aspects to "correctness" and the protocol state space is very large. Furthermore, a simple correctness property, such as "a load from an address always returns the value of the last write," specified at the architectural level is often extremely difficult to verify in the implementation. Moreover, discovering errors in the pre-silicon stages of the development is paramount to the both the budgetary and schedule success of the project. Our approach is to formally model the protocol using the Murφ verification system and prove that a set of well-defined, fundamental properties hold over the state space. Then, by capturing *witness strings* from a depth-first search of the protocol state space and executing the traces on the Verilog RTL logic simulation we show that the implementation matches the formal specification. While the witness strings approach has been effective at uncovering errors in the Cray X1 cache coherence protocol [1] both in the high-level protocol specification and its RTL implementation, it does suffer from computational complexity when executing the witness string on the RTL logic simulator. For instance, the model checker explores 214 million states in 208 hours which averages to about 287 states per second. However, the logic simulator is only capable of exploring about 10 states per second which would require weeks of compute time. While it is tractable to execute all the witness strings on the RTL logic simulator, it is not very *practical* within the constraints of a project development schedule.

Each witness string is on the order of a few thousand states long, and the Murφ model checker will yield thousands of strings that will be executed by the RTL logic simulator. It is reasonable to ask, if some witness strings are more efficient than others at exposing errors? If so, we would like to prioritize how we execute the witness strings on the RTL so that the "best" witness strings are selected first. This research focuses on search heuristics that will guide the depth-first search of the model checker to produce witness strings that are more efficient at uncovering errors. To evaluate the efficacy of each heuristic, we measure how many states are explored before the error is discovered. We chose cache coherence protocols from the Cray X1 and Stanford DASH multiprocessors as an experimental substrate. Section 2 gives a framework for the witness strings approach and motivates the case for search heuristics. We evaluate three heuristics: hamming, cache-score and min-max-predict which are discussed in Section 3. Then, in Section 4 we provide the experimental setup used for evaluating the efficacy of each heuristic. The experimental results are shown in Section 5, with related work in Section 6. Finally we draw some conclusions in Section 7.

2 Witness strings methodology

Model checking is a method for verifying a system comprised of concurrent finite-state machines. It entails searching the state space to ensure that a set of properties hold as each state is visited. Since the model itself is finite, the reachable state space will also be finite, and therefore the formal verification is guaranteed to eventually finish. However, the search method will explore all possible interleavings of concurrent transitions. As the size of the formal model grows, the reachable state space grows exponentially thereby creating what is commonly referred to as the state space explosion problem.

We begin by formally defining an FSM by the 5-tuple:

$$FSM = (Q, \Sigma, \delta, q_o, F) \quad (1)$$

where Q is a finite set of states, Σ is a finite *input alphabet*, $q_o \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping $Q \times \Sigma$ to Q. The transition function, $\delta(q, x)$ takes as its argument the current state, $q \in Q$, and an input symbol, $x \in \Sigma$, to produce a new state Q. We extend the definition of δ slightly, so that the transition function may produce zero or more *actions*, $z \leftarrow \delta(q, x)$ based on the current state q and input message x.

2.1 Tabular Specification

The cache coherence protocol is represented as a set of tables describing the current state, incoming message, next state and action (Table 1). For instance, consider the cache coherence protocol specification for the L2 cache of the Cray X1. If the current state is Invalid and a Read request from the processor is received, then the current state transitions to the Pending state and the L2 cache sends an MRead request to the memory directory (M) chip. Actions are separated by a semi-colon in the table, and all actions are considered to occur atomically. That is, they give the appearance of occurring simultaneously.

Table 1: A snippet from the Cray X1 L2 cache coherence protocol specification.

Current State	*Incoming Command*	*Next State*	*Action*
Invalid	Read	Pending	M(MRead)
Invalid	ReadMod	Pending	M(MReadMod) ; increment(wc)
Invalid	ReadNA	Invalid	M(MGet)
Invalid	VWrite	Pending	M(MReadMod) ; increment(wc)
Invalid	VWriteNA	Invalid	NOP()
Invalid	VWriteData	Invalid	M(MPut) ; increment(wc)
Invalid	ReadSharedResp	*err*	ERROR()
Invalid	ReadExclResp	*err*	ERROR()
Invalid	FlushReq	Invalid	M(MFlushAck)
...	...	...	...

2.2 Verification process and flow

The Murφ formal verification model is generated by a program (`tbl2m`) that converts the tabular specification described in Table 1 into Murφ source code. Then, the Murφ source is compiled with the `mu` compiler producing a verifier. To generate witness strings, the verifier is executed and the output is piped into the `ws` program that disambiguates node numbers and addresses which are lost as a result of symmetry reduction [2] in the verifier. The output from the `ws` program is converted into stimulus for the logic simulator using the `ws2r` which converts from the witness strings into Raven [3] source code. The Raven source code is then compiled using `gcc` and executed using the Synopsys VCS RTL logic simulator.

2.3 Formal Language Framework

Our approach has its genesis in the theory of NP-completeness taken from computational complexity theory. Briefly, all problems that belong to the class NP possess the attribute of polynomial-time verifiability. That is, if we magically knew the solution to an NP-hard problem we could then verify (i.e. check the correctness of) the solution in polynomial time. We first construct a formal model of the cache coherence protocol. Then, by using formal verification (model checking) to produce a set of verification certificates called *witness strings* and executing them individually on the Verilog RTL simulation we hope to establish that the implementation is functionally consistent with the formal model. To show how this approach works, we first describe the formal model of computation based on a k-tape nondeterministic Turing machine (NTM) and how the coherence protocol is executed under this model. Then, we describe the formal language that this model accepts.

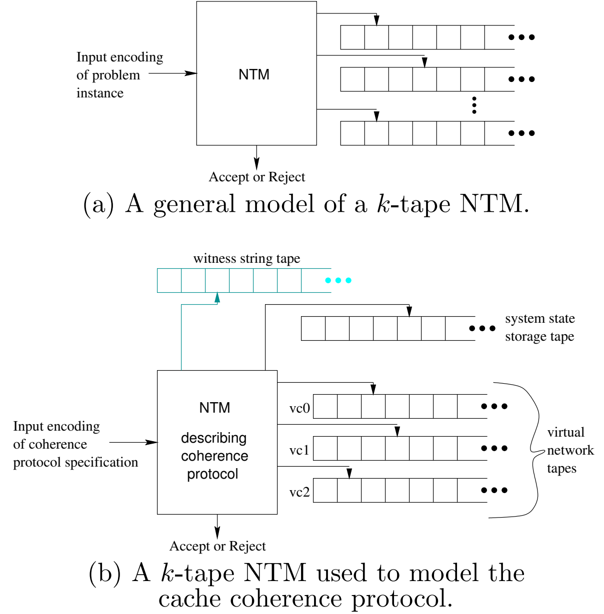
The Turing machine model [4, 5] is commonly used in complexity theory to characterize the run-time complexity of an algorithm. We describe a variant of this model of computation called a k-tape nondeterministic Turing machine (NTM). To be specific, the NTM is an acceptor machine which decides (accepts or rejects) if the input is an acceptable solution. The NTM consists of some finite control units and k tapes that are used for storage (Figure 1). The NTM is defined as the 6-tuple

$$NTM = (Q, \Sigma, \Gamma, \delta, q_o, F) \quad (2)$$

where Q is a finite set of states, Σ is the input alphabet, Γ is the *tape alphabet*, which includes a special *blank* ($\Box$) symbol, with $\Box \in \Gamma$ and $\Sigma \subseteq \Gamma$. The transition function, δ, is augmented with {L, R} operators to control the movement of the tape head to the left or right, respectively.

The *configuration*, C_i, of the NTM describes the current state, tape contents, and head location. The initial configuration, C_0 is given by the initial state q_0, with a $\Box$ symbol on each tape cell, and the tape head positioned at the left-most cell of the tape. The NTM operates by applying *rules* that consist of a sequence of atomic actions described by the

Figure 1: A formal model of computation based on a k-tape nondeterministic Turing machine (NTM)



(a) A general model of a k-tape NTM.

(b) A k-tape NTM used to model the cache coherence protocol.

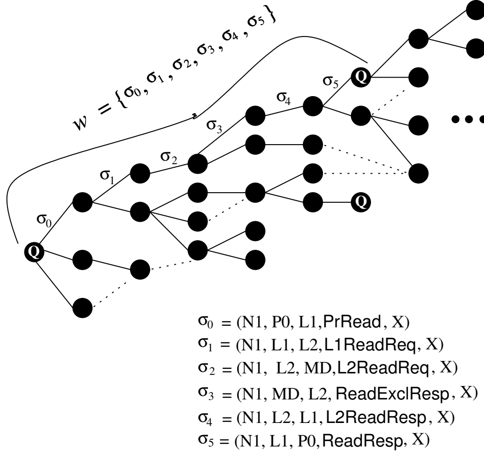
transition function, where each configuration C_i yields a new configuration C_{i+1}. Intuitively, each configuration C_i represents the *global system state* of the system being modeled.

2.3.1 Formal model

The formal model of the system being verified, M, is an NTM consisting of all the cooperating FSMs that describe the coherence protocol in addition to an FSM that acts as the load-store unit generating all possible processor requests. Let n be the number of nodes being analyzed. Each node has 1 bit of local memory. Each cache has a single line with only a single-bit data value, requiring $lg(n)$ tag bits to represent the cache tag. A storage tape is used to model each virtual channel of the interconnection network (Figure 1b). The FSMs are processes that execute when their *rule condition* is satisfied. For example, the rule condition for the FSM_VC0 state machine is "wait for an incoming message on vc0 tape head." When the condition is satisfied, the rule fires causing the incoming message to be consumed and a $\Box$ symbol to be written on the tape. The tape head then is moved one position to the right (using the R tape head operator). Any messages that are produced from this rule firing are written to the appropriate virtual channel tapes.

At each configuration, the set of eligible rules are those in which the rule predicate is satisfied. We use the notation $C_i \xrightarrow{r} C_{i+1}$ to signify that applying rule r while in configuration C_i yields a new configuration C_{i+1}. After applying each rule, we inspect the storage tape containing all previously explored configurations. If the new configuration is unique, we add the configuration C_{i+1} to the storage tape and move the head one position to the right. However, if applying the rule does not yield a new state (i.e. $\exists j \in \{0..i\} : C_j \equiv C_{i+1}$) then we choose another rule and continue to execute until we produce a new configuration or we run out of eligible rules. The eligible rules can be exhausted in three possible ways: 1) the search has reached a leaf node, 2) we have searched all possible configurations, or 3) the verifier has deadlocked. We distinguish the deadlock case by noting that, if a deadlock occurs, the NTM will have unserviced symbols (messages) on the virtual channel tapes. If all the eligible rules are exhausted, and we have not deadlocked, we move the storage tape head one position to the left assign the current configuration C_i to the contents stored at the storage tape head. This process is repeated until the initial configuration C_0 is removed from the storage tape after having exhausted all its eligible rules, in which case the NTM halts.

Figure 2: A collection of symbols form a word, w, in the language $\mathcal{L}$(M).



It is useful to note that the NTM is enumerating all the reachable states of the coherence protocol in a depth-first manner. As each new configuration is produced the invariants are checked to ensure the correctness properties hold. If any of the correctness properties are violated, the NTM will halt in a rejecting configuration with the tape contents providing evidence of the failure. As the formal model executes, it produces a computation tree (Figure 2) where each node in the tree is a new configuration of the NTM and each arc represents the rule firing. The symbols $\sigma_1, \sigma_2, \ldots, \sigma_n$ represent rule firings, where $C_i \xrightarrow{\sigma_i} C_{i+1}$. The set of symbols $\{\sigma_0, \sigma_1, \ldots, \sigma_i\}$ that traces a path from the starting configuration C_0 to a leaf node is called a *witness string* for the execution of the formal model. In Figure 2, the dotted arcs represent a rule firing that does not produce a new (unique) state and is therefore not considered part of the computation tree. The σ symbols are recorded on the witness string tape of the NTM as each rule is fired.

Each σ symbol is a 5-tuple describing the rule firing as:

$$\sigma_i = (n, src, dest, cmd, addr)$$

where n is the node identifier, *src* is the source component, *dest* is the destination component, *cmd* is the command (message) type, and *addr* is the address. We represent the address symbolically so that, for instance, a three node system would have three addresses: X, Y, and Z. An example of a witness symbol is (N1, P0, L1, PrRead, X) which is a processor request from P0 to the L1 cache controller to perform a PrRead of address X. A simple memory transaction will consist of the several symbols that witness the memory reference as it propagates through the memory hierarchy. A *word* in the witness string is a collection of symbols separated by *quiescent* configurations, as shown in Figure 2. A quiescent configuration is one in which the virtual channel tapes contain only blanks ($\Box$) and the state at each level of the memory hierarchy is also quiescent (i.e. the state is something other than Pending). Intuitively, this means that all outstanding requests have been satisfied and there are no messages in-flight in the virtual network. Formally, a word $w = \{\sigma : C_i \xrightarrow{\sigma} C_j\}$, where C_i and C_j are quiescent configurations or C_j is a terminal configuration (leaf node).

2.3.2 Formal model equivalence

Given our formal model, M_f, of the cache coherence protocol we define a witness string as a collection of words from the initial configuration, C_0, to a terminal configuration (leaf node), C_i. The language accepted by the formal model, $\mathcal{L}(M_f)$, is the concatenation of all witness strings. If two models accept the same language then they are *equivalent* [5]. If the model, M_ℓ of the RTL implementation (i.e. the logic simulation) accepts the same language then $M_f \equiv M_\ell$. We use the witness strings as a verification certificate between the formal verifier and the RTL logic implementation. Showing that both the formal verification model and the RTL implementation accept the same language, namely the language described by the collection of witness strings, gives confidence in the formal verification matching the implementation.

3 Search Heuristics

We propose several search heuristics to optimize model checking of a cache coherence protocol. In particular, the heuristics seek to improve depth-first search by guiding the search process toward a portion of the protocol state space that is more likely to contain an error. Then, using the optimized search from the model checker we are able to produce witness strings that are more efficient, capable of finding errors in less time, than depth-first search. In the presence of computational complexity, faced with the daunting task of executing the voluminous numbers of witness strings on the logic simulator, the witness strings produced by the optimized model checker provide a complexity-effective verification of the protocol implementation. We evaluate the search heuristics on the Cray X1 and Stanford DASH.

3.1 Hamming distance

The hamming distance [6] between two states, s_1 and s_2, is defined as the number of bits by which s_1 and s_2 differ. Two heuristics max-hamming and min-hamming select the rule with either the largest or smallest hamming distance, respectively. Intuitively, the max-hamming heuristic will choose the next state that is most different from the current state, whereas the min-hamming heuristic chooses the next state that is least different. In Murφ each state is stored as a large bit-vector. We simply loop through the bit-vector and compare $s_1 \rightarrow$bits$[i]$ to $s_2 \rightarrow$bits$[i]$ incrementing a counter each time a difference is detected. Thus the overhead in computing the hamming distance is linear with respect to the size of each state.

3.2 Cache Score

It is common that a small number of control bits can dominate the circuit functionality. For instance, the cache *state* and *tag* bits dominate the functional behavior for the L2 cache coherence controller. The cache-score heuristic uses a subset of the state information to determine the best rule to fire. For the Cray X1 protocol, the cache-score evaluation function inspects the internal state variables of the ws-Murφ verifier and returns a score, $s = [0{:}4]$. For example, for each memory directory (MD) component in the model, we score it according to the following code.

```
int md_score(int n) { // n is the node number
int s=0;
if(Node[n].Directory.State != Noncached &&
   Node[n].Directory.State != Shared    &&
   Node[n].Directory.State != Exclusive)
      s = 1 ;
return s ;
}
```

A similar function e_score( ) is used to score the L2 cache. The md_score( ) and e_score( ) functions take the node number as an argument. So, the aggregate score for each state is as follows:
score = e_score(1) + e_score(2) + md_score(1) + md_score(2); A rule which produces a next state with a higher score is favored over a rule the produces a lower score. The score gives a metric of the amount of concurrent coherence traffic. Intuitively, a higher score means there are more outstanding coherent memory references and therefore more likely to produce an error.

3.3 Min-Max-Predict

Yang and Dill [7] used the minimum hamming distance as a search heuristic with the hope that states with very few bits differing from the error state will require fewer cycles to reach the target. While our reasoning is that by choosing the next state via *maximum* hamming distance the search will

move toward the error state. To exploit the strengths of both minimum and maximum hamming, we arrived at our min-max-predict heuristic by combining the min-hamming and max-hamming heuristics with the scoring function from cache-score and a saturating counter to predict the best next state based on the current state. The semantics of the next state predictor is very similar to the way a branch predictor predicts the next program counter and captures the notion of hysteresis in a program control flow.

The heuristic works by first determining a score for the current state in the range [0:n]. If the score $< n/2$, then we increment the counter, otherwise we decrement the counter. We make sure the count never exceeds the bounds of the 3-bit counter by first checking if count <8 before incrementing it, and checking if count > 0 before decrementing the counter. Then, after we fire a rule we compute the hamming distance from the current state to the new state. We then use the hamming distance to set the min_rule and max_rule values that are used to evaluate the min-hamming and max-hamming heuristcs, respectively. The value from min_rule or max_rule is assigned to the variable best_rule depending on the current counter value. If count <4 then best_rule = max_rule, otherwise best_rule = min_rule. The up-down 3-bit saturating counter is used to steer the search toward the target error.

4 Experimental setup

We chose the Cray X1 and Stanford DASH cache coherence protocols as an experimental substrate. We embed errors into each protocol and use the model checker to search for the known error. This process of hiding and seeking memory coherence errors is repeated for six different errors on each protocol. While the X1 and DASH protocols are both directory-based [8] coherence protocols, they are largely dissimilar in other respects. For example, the X1 protocol is a blocking protocol and DASH uses NAKing (negative acknowledgments) to defer incoming requests to a pending cache line. The DASH protocol is also the predecessor to the SGI Origin2000 [9] cache coherence protocol.

The Murφ formal verification model of the DASH protocol has a total reachable state space of 10466 states where the size of each state is 4912 bits. The Cray X1 cache coherence protocol has a much larger state space with 214 million reachable states where the size of each state is 1664 bits. Each error is listed below with the invariant that failed.

4.1 Invariants

Model checking is a technique that exhaustively searches the states space of a concurrent system to show that certain properties hold. Safety properties are simple invariants expressed in first-order logic that, in essence, ensure that "something bad never happens." The invariants we checked are common to all cache coherence protocols and ensure such properties as exclusive write permission to a cache line, and data consistency among others. As an example, consider the "single writer" property for the Cray X1 cache coherence protocol [1].

> Two different caches, **p** and **q**, should never have write access to the same address, **a**, at the same time.
>
> $$\forall_a \forall_p \forall_q \, \mathbf{IsDirty}(a,p) \wedge q \neq p \Rightarrow \neg \mathbf{IsDirty}(a,q)$$

The single writer property is written in Murφ as:

```
Invariant "Single Writer."
Forall a : Address Do
 Forall p : NodeID Do
  Forall q : NodeID Do
    (p != q) & IsDirty(a, p) -> ! IsDirty(a, q)
  End
 End
End ;
```

The DASH protocol verifier has an equivalent safety property that detects multiple caches with write permission. There are other invariants that check for data consistency, unexpected messages, etc.

4.2 Model checking

We used the Murφ verification system [10] to check our formal model. Murφ uses explicit state enumeration to exhaustively search the reachable state space of a system of interacting finite-state machines. In addition to deadlock freedom, the verifier checks that a set of invariants are satisfied at each explored state. We did, however, make small modifications to the original Murφ source code to allow us to capture witness strings from the verifier. The new verifier, called ws-Murφ, has minor modifications to several functions to capture witness strings and allow the addition of search heuristics.

To capture the witness strings, we added two lines of code to the original Murφ function verify_dfs in the mu_system.C file. The first modification prints a special token before each rule is fired. The second modification prints a message when the depth-first search backtracks. Then, each cache controller rule prints out the witness symbol as it executes, as shown in Figure 3. The witness symbols are converted into a Raven [3] diagnostic program that can be compiled and run with the Synopsys VCS simulator. The ws-Murφ verifier will generate millions of lines of C++ code to run on the logic simulator. This process would benefit tremendously from a search heuristic that yielded witness strings that are better at discovering errors than those generated from a depth-first search.

To implement search heuristics in ws-Murφ we added a new function AllNextStates_BestNS which chooses the rule to fire based on our heuristic, instead of the SeqNextState function. We also modified functions NextState, was_present, and simple_was_present to include a boolean flag indicating that we are searching a state that is being evaluated by the heuristic and therefore should not be added it to the hash table. For each state the heuristic returns a score that is used as a metric to determine the best rule to execute and produce the next state. The search process is guided by the heuristic toward a potential error.

4.3 Cray X1 embedded errors

Errors for the Cray X1 were chosen to provide a variety of errors with different invariant failures. Error1 violates the single writer invariant, Errors2-4 generate protocol errors, Error5 creates deadlock, and Error6 violates the data coherence invariant. Errors embedded in the X1 protocol are roughly divided into two categories: quiescent and transient state errors. That is, does the error occur when the L2 cache or memory directory is in a quiescent (not pending) or transient (pending) state.

Error1 L2(WaitForVData)←FwdRead
: The L2 receives a FwdRead request while in a pending state waiting for vector write data. The FwdRead causes a SupplyDirtyInv reply to be sent to the requester and should transition to the Invalid state. Instead, the L2 state remains in the WaitForVData state.
Invariant violated: Single Writer

Error2 L2(ExClean)←FwdRead
: The L2 receives a FwdRead request while in the ExClean

Figure 3: Witness strings from the ws-Murφ verifier.

```
. . .
-------------------------------------------------
E2(X:ExClean)<---FlushReq(X) [C=0] on vc1 from M1
Quiescent: 0
    E2 sending PInvalidate(X) [C=0]  to Proc_1
    E2 sending MSupplyInv(X) [C=0]  to M1 on vc2
-------------------------------------------------
M1(PendFwd)<---MSupplyInv(X) [C=0] on vc2 from E2
Quiescent: 0
This state has been examined, try another rule.
-------------------------------------------------
E2(?:Invalid)<---VWrite(Y) [C=0] on vc0 from P1
Quiescent: 0
    E2 sending MReadMod(Y) [C=0]  to M2 on vc0
incrementing write counter
-------------------------------------------------
M1(PendFwd)<---MSupplyInv(X) [C=0] on vc2 from E2
Quiescent: 0
-------------------------------------------------
. . .
```

Table 2: Results for X1 and DASH. The entries in the table are the number of states explored by ws-Murφ.

	DFS	BFS	Hamming Min	Hamming Max	Cache Score Min	Cache Score Max	Min-Max Predict
Error1	753	84915	23885	64	9218	265	35
Error2	483	88472	26	1704	2751	1784	93
Error3	1037	53808	2163	537	7293	693	751
Error4	101424	53815	223	30314	46910	109103	2118
Error5	101413	770162	268	78456	46910	109093	2180
Error6	101423	45320	223	30302	46901	109103	1787

(a) Results for the Cray X1. The min-max-predict heuristic performs better than DFS or BFS for all the error cases. The largest improvement reduces the search states from 101423 to only 1787 states.

	DFS	BFS	Hamming Min	Hamming Max	Cache Score Min	Cache Score Max	Min-Max Predict
Error1	4719	3742	9093	4958	932	8553	2411
Error2	421	715	25	437	54	409	124
Error3	609	1034	3794	745	955	399	584
Error4	533	418	10265	504	924	78	410
Error5	773	1062	3794	1096	955	582	584
Error6	274	465	120	199	62	339	199

(b) Results for the Stanford DASH. The min-max-predict heuristic performs better than DFS or BFS for all the error cases. Although the improvement is not as drastic as the results for the Cray X1 protocol.

state. The L2 responds with a SupplyDirtySh but never transitions from ExClean, to a shared state, ShClean.
Invariant violated: Protocol Error

Error3 L2(Pending)←Inval
The L2 cache receives an Inval message to invalidate the cache line, and erroneously sends a FlushAck instead of an InvalAck message.
Invariant violated: Protocol Error

Error4 MD(Shared)←Drop
The memory directory (MD) receives an L2 cache eviction message, Drop. The MD erroneously transitions to the Exclusive state instead of staying in the Shared state.
Invariant violated: Protocol Error

Error5 MD(Shared)←Read
The MD receives a Read request from the L2 cache, but never adds the cache to its sharing vector so the MD is not correctly tracking caches with a shared copy.
Invariant violated: Deadlock

Error6 MD(PendDrop)←Drop
The MD has detected that an L2 cache has re-requested a shared cache line and thus an L2 eviction notice must be in-flight. However, the MD erroneously does not toggle the bit in the sharing vector when it receives the Drop message.
Invariant violated: Data Coherence

4.4 Stanford DASH embedded errors

DASH has three types of requesters: the *local* or *remote* cluster, and *home* cluster, as well as two classes of memory references to local or remote memory. Errors were implanted by inspecting state diagrams of the protocol given in [8] and embedding errors that vary the failing invariant. Error1 is an error that was re-discovered with Murφ after it was originally uncovered after substantial amounts of simulation.

Error1 Handle read exclusive request to home
No invalidation is sent to the master copy requesting cluster, which has already invalidated the cache.
Invariant violated: Consistency of data

Error2 Send reply message instead of NAK
Instead of a negative acknowledgement (NAK) message, an acknowledgement (ACK) message is sent in reply.
Invariant violated: Writeback from non-dirty remote

Error3 Handle read request to remote cluster
Instead of changing the cache block in the remote cluster to be locally shared after sending the data block to the requesting remote cluster, the cache block remains locally exclusive.
Invariant violated: Writeback from non-dirty remote

Error4 Handle read exclusive request to remote cluster
Instead of changing the cache block in the remote cluster to be not locally cached, the cache block remains locally exclusive.
Invariant violated: Only a single master copy exists

Error5 Handle DMA read request to remote cluster
The dirty cache block changes to be locally shared instead of staying dirty after supplying the data.
Invariant violated: Writeback for non-dirty rmt DMA

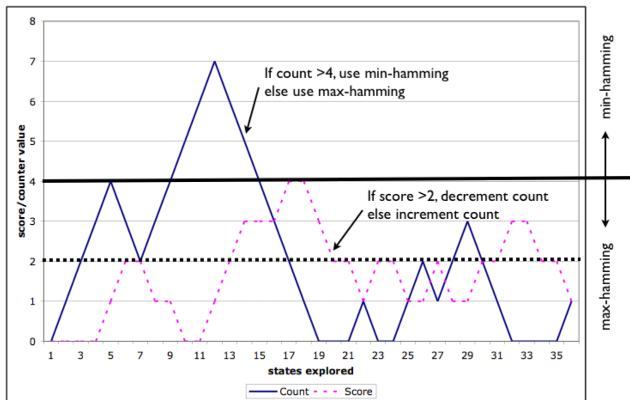
Error6 Handle invalidate request to remote cluster
After collecting all the invalidation acknowledgements, the request entry in the Remote Address Cache (RAC) should transition to the invalid state. Instead, request entry is preserved and the state remains as waiting for a read reply.
Invariant violated: Condition for existence of master copy

5 Results

A summary of our experimental results is given in Table 2. It is generally believed that a breadth-first search (BFS) is more efficient than depth-first search (DFS) because a BFS produces a shorter error trace when an invariant fails [10]. Interestingly, for both protocols a depth-first search (DFS) outperforms breadth-first search (BFS) on eight of the twelve error cases. Errors 4, 5, and 6 for the Cray X1, see Table 2(a), are quite deeply ensconced in the state space, having to explore in excess of 100 thousand states before the targeted error was discovered! These three errors had one thing in common, the error effected only the memory directory state or sharing vector and not the L2 cache state or messages exchanged. Even though the error occurs very early in the search process, *discovery* took much longer. On several of the errors in both protocols, the min-hamming heuristic proved to be very fruitful. However, it did significantly worse than DFS on several cases. Interestingly, when min-hamming performed poorly, the max-hamming heuristic always performed well, usually significantly better than DFS. Taken by themselves, the min-hamming and max-hamming heuristics do not perform consistently across the broad set of protocol errors.

The cache-score heuristic uses the cache state and memory directory state to guide the search. By favoring states with

Figure 4: A plot of counter and score values as the search for the Cray X1 embedded Error1 progresses from the initial state to error discovery.



more outstanding requests we hope to guide the search toward states that are more likely to have errors. Unfortunately, this heuristic does not perform consistently because it biases the search toward states that are pending without allowing the rules that would resolve the pending state to be fired until much later in the search. The state of a cache memory will waver between quiescent, pending, quiescent, pending, quiescent, etc. This is because new requests that cause an eviction will move the state from quiescent to pending, and likewise responses from the memory controller will move the state from pending to quiescent again.

The min-max-predict heuristic chooses either the min-hamming or max-hamming heuristic depending on a prediction from a small saturating counter. It allows the search to move from a region of the graph to another and switch heuristics to best suit the more local search criteria based on the scoring function from the cache-score heuristic. In effect, it uses the cache state information to make a locally informed search decision in hopes to yield an effective global search result. Figure 4 shows the counter value and score from the initial state to the error state. From Table 2(a) and (b) we see that min-max-predict is consistently vastly better than BFS and DFS. This improvement is often several orders of magnitude better for the Cray X1 protocol! Contrary to the findings of Yang and Dill [7], we found that min-hamming performed better on only two of the six errors for the DASH protocol.

For the min-max-predict heuristic we experimented with 2-bit, 3-bit and 4-bit counters in the predictor. We found that a 3-bit counter worked best for the Cray X1 protocol and a 4-bit counter worked best for the DASH protocol. Perhaps a small-valued counter in the range [0,12], for instance, would yield a nice compromise for both protocols. We believe the min-max-predict search heuristic would be suitable for verifying most cache coherence protocols and could be applied to other domains.

6 Related Work

Yang and Dill [7] examined the benefit of minimum hamming distance to improve a bread-first search in the hope that states with very few bits differeing from the target will require very few steps to reach the error. They use a technique called "target enlargement" to expand the size of the error states to all states that are 1 search step away from the error state. They concluded that minimum hamming distance could reduce the number of states explored, however its performance was very inconsistent across different designs. They also studied a "guideposts" heuristic similar in concept to our cache-score heuristic. They use target enlargement in combination with guideposts to further reduce the states searched. All the heuristics were compared to a baseline BFS. Further, they considered only a single error from five designs: four from the memory controller of the Stanford FLASH multiprocessor and one from the link-layer communication protocol used by the Sun S3.mp multiprocessor. Yang and Dill explored the minimum hamming distance heuristic, but did not address any possible merits of maximum hamming distance. Our study shows that most of the time DFS performed better than BFS.

Yuan, et. al. [11] use retrograde analysis to combine symbolic verification with simulation. They found hamming distance to be a useful heuristic to reduce the number of simulation trials needed to reach an enlarged set of errors called the pre-image of the error. Our approach does not require computing the enlarged target error states. Our approach uses model checking as opposed to symbolic verification.

7 Conclusions

A detailed formal verification of a highly concurrent cache coherence protocol, such as that used by the Cray X1, yields a state space with 214 million reachable states. We constructed a formal verifier called ws-Murφ which can explore approximately 300 states per second, whereas an RTL logic simulation of the coherence protocol running in Synopsys VCS can execute about 10 states per second. We evaluated several search heuristics with the goal of reducing the number of explored states compared to depth-first search (DFS) and breadth-first search (BFS). Our min-max-predict consistently performs better than DFS and BFS. We are most concerned with improved performance compared to DFS since this will produce witness strings that are executed with a "best-first" policy. Being able to cull the most efficient witness strings from a voluminous protocol state space can reduce the error discovery time in simulation by several hours. We suspect the min-max-predict search heuristic would be suitable for verifying most cache coherence protocols and could be applied to other domains.

References

[1] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proceedings o fthe 17th International Parallel and Distributed Processing Symposium (IPDPS03)*, April 2003.

[2] C. N. Ip and D. L. Dill. Better verification through symmetry. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112, Amsterdam, The Netherlands, 1993. North-Holland.

[3] Dennis Abts and Mike Roberts. Verifying large-scale multiprocessors using an abstract verication environment abstract verification environment. In *Proceedings of the 36th Design Automation Conference (DAC99)*, pages 163–168, June 1999.

[4] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, pages 146–176. Addison Wesley, 1979.

[5] M. Sipser. *Introduction to the Theory of Computation*, pages 53–63, 54, 125–147. PWS Publishing, 1997.

[6] R. W. Hamming. Error detecting and correcting codes. Technical Report Vol 29, Bell Laboratories Technical Journal, pp. 147-160, 1950.

[7] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Annual Design Automation Conference (DAC98)*, June 1998.

[8] D. E. Lenoski. The directory-based cache coherence protocol for the DASH multiprocessor. *Proc of the 17th Annual Int. Symposium on Computer Architecture*, pages 148–159, June 1990.

[9] James Laudon and Daniel Lenoski. The SGI origin: A cc-NUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New York, 2–4 1997. ACM Press.

[10] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, 1992. IEEE Computer Society Press.

[11] Jun Yuan, Jian Shen, Jacob Abraham, and Adnan Aziz. On combining formal and informal verification. In *Proceedings of 1997 Conference on Computer Aided Verification (CAV1997)*, pages 376–387, 1997.