HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols

Nicolai Oswald
The University of Edinburgh
nicolai.oswald@ed.ac.uk

Vijay Nagarajan
The University of Edinburgh
vijay.nagarajan@ed.ac.uk

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

Abstract—**We present HieraGen, a new tool for automatically generating hierarchical cache coherence protocols. HieraGen's inputs are the simple, atomic, stable state protocols for each level of the hierarchy. HieraGen's output is a highly concurrent hierarchical protocol, in the form of the finite state machines for all of the cache and directory controllers. HieraGen thus reduces the complexity that architects face, by offloading the challenging tasks of composing protocols and managing concurrency. Experiments show that HieraGen can automatically generate correct-by-construction MOESI family of hierarchical protocols with dozens of states and hundreds of transitions. We have verified all of the generated protocols for safety and deadlock freedom using a model checker.**

I. Introduction

Designing a cache coherence protocol for a multicore processor is a challenging task, yet new protocols must frequently be designed for new multicore processors. As processor designs change—with the addition of more cores or different types of cores, or with different expected communication patterns—there are incentives to create new coherence protocols to suit these changes. Even if a new protocol is not a radical departure from previous protocols, designing it and validating it are arduous, bug-prone processes.

One source of protocol design complexity is concurrency. If one considers only atomic protocols, such as those sometimes found in textbooks, then protocol design seems fairly simple. These atomic protocols, also known as stable state protocols (SSPs), have only a handful of stable coherence states (e.g., MESI), and have easily understood state transition diagrams. However, modern protocols are highly concurrent, so as to achieve as much performance as possible. Many transactions can be in progress at once and it is the races among concurrent transactions to a single block that lead to design complexity. Sorin et al. [1] present concurrent directory protocols with dozens of transient states and significant complexity, and industrial coherence protocols can be even more complicated.

To overcome the design complexity of coherence protocols, a recent design automation scheme called ProtoGen [2] converts a SSP into a highly concurrent protocol design. The designer need only reason about the SSP and not consider the transient states and extra transitions that are needed to accommodate concurrency. ProtoGen was shown to be effective in creating concurrent *flat directory protocols*, in which a single directory—perhaps colocated with a shared cache—communicates directly with all of the cores and their private cache hierarchies, as shown in Figure 1(a).

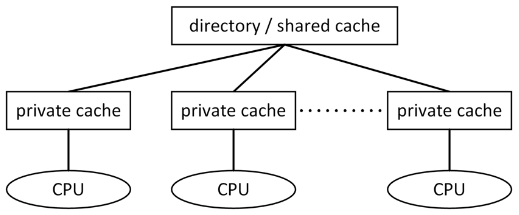
ProtoGen is a step towards automation, but it is restricted to a very narrow system and protocol model. While we expect directory-like coherence to persist, there are several trends pushing industry away from flat protocols and towards protocols with hierarchy. Hierarchy is a time-tested design strategy for scalable systems [3]–[10], and it is an attractive approach to multicore processor design as the number of cores continues to increase. Hierarchy can also enable coherence protocols that are more scalable [11]. Figures 1(b) and 1(c) show two hierarchical system models with hierarchical directories (and hierarchical shared caches).

While hierarchy has many desirable features, it also greatly complicates the design of the coherence protocol. There are more states, more transitions, and more possible concurrency. Crucially, communication between levels of the hierarchy must preserve coherence invariants. In addition to the design complexity, verification is also more challenging with hierarchy, due primarily to the much larger state space [9], [12].

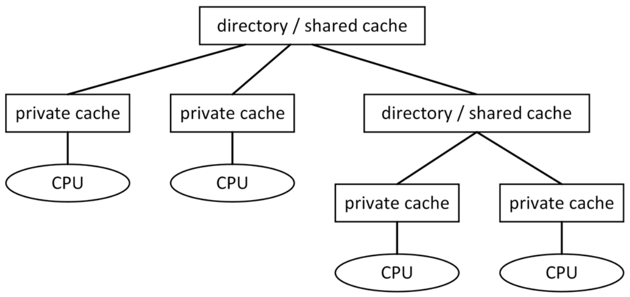
To sidestep the challenges in designing (and verifying) hierarchical coherence protocols, we introduce HieraGen[1], a design automation tool for generating correct-by-construction hierarchical protocols.

The user inputs the SSPs of each level independently. For example, as shown in Figure 1(d), the user would input two SSPs (shown in different colors): one for the protocol of the subtree in the bottom right (oblivious to the higher level) and one for the protocol of the higher level (oblivious to the subtree in the bottom right). The user would also specify the point(s) at which the protocols are connected, e.g., that the subtree protocol is a node in the higher level protocol. (For clarity, we refer to a core with its private cache(s) as a core/cache node and a directory with a collocated, optional shared cache as a directory/cache node, and we use standard tree terminology (root, parent, child) to identify nodes in the hierarchy). Thus, the higher level SSP would include only the specifications of the root directory/cache and the core/cache nodes that are its children, as in the specification of a flat protocol; it would *not* include any information about the possibility of a child that is an integrated directory/cache node.

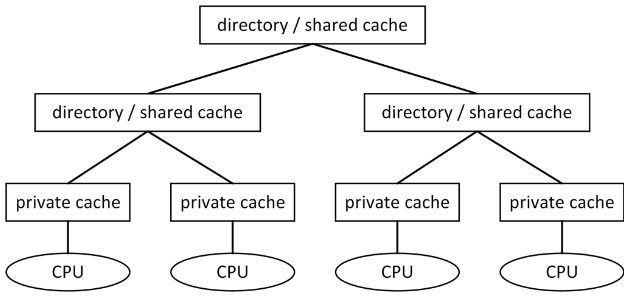
[1]https://github.com/icsa-caps/HieraGen



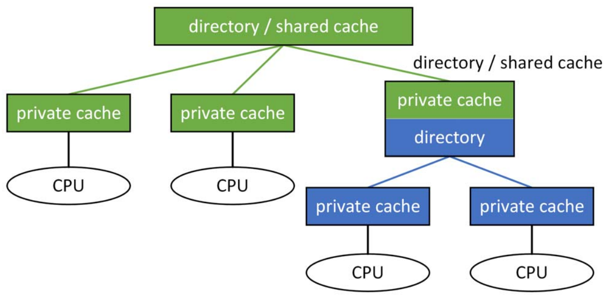
(a) flat directory



(b) hierarchical directory



(c) multi-hierarchical directory



(d) HieraGen composition of two SSPs (in different colors)

Fig. 1: System Models

Given these inputs, HieraGen produces the design of the complete and concurrent hierarchical protocol, i.e., Figure1b. The output is a finite state machine that is currently in the language of the Murϕ model checker [13], to facilitate verification, but could easily be transformed to other finite state machine representations (e.g., Verilog, VHDL, or SLICC).

Accommodating hierarchy introduces a key challenge beyond what ProtoGen addresses: HieraGen must create intermediate (non-root) directory/cache nodes that were not completely specified by the user; as shown in Figure 1d, HieraGen must compose the cache from the higher level (in green) with the directory from the lower level (in blue) to produce the intermediate directory/cache node.

HieraGen addresses this challenge by automatically encapsulating higher-level coherence actions within lower-level coherence transactions (and vice versa), such that coherence invariants are enforced globally.

HieraGen must also implement concurrency in a hierarchical system that can have multiple coherence transaction serialization points (e.g., directories). We make the observation that in a hierarchical SSP that correctly enforces coherence globally, any two racing coherence transactions will serialize at exactly one of the directories. This key invariant allows us to leverage ProtoGen for generating concurrency.

HieraGen is currently limited to coherence protocols that enforce the “single-writer, multiple reader” (SWMR) coherence invariant. It is also limited to inclusive cache hierarchies. Finally, the protocols generated by HieraGen do not allow for direct communication between nodes of different hierarchy levels.

In summary, the contributions of this paper are as follows.

- We present HieraGen, the first automated tool for taking SSP specifications of coherence protocols of each level of a hierarchical system, and generating the complete and concurrent hierarchical protocol, while preserving safety and preventing deadlocks.
- We have used HieraGen to generate high-performance hierarchical protocols with the standard MOESI coherence states.
- We have verified the generated protocols for safety and deadlock freedom using the Murϕ model checker.

II. Background

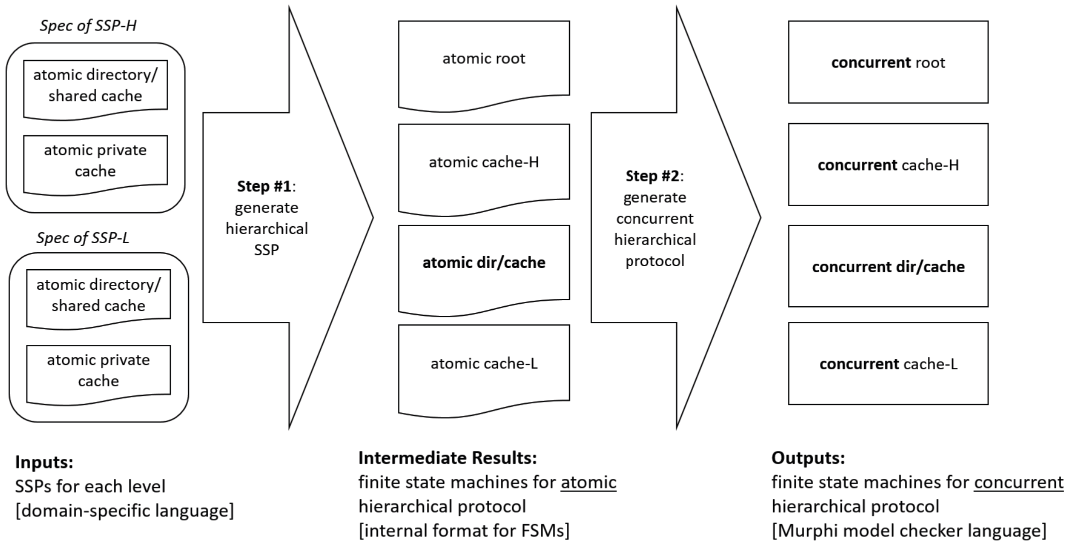
In this section we provide a brief background on coherence protocols. We also provide a brief overview of ProtoGen, which HieraGen leverages for generating concurrency.

A. Coherence Protocols

An important class of coherence protocols, including most multicore coherence protocols, satisfy the Single-Write-Multiple-Reader (SWMR) invariant [1]. For any given memory location, at any given time, there is either a single core that may write to it or some number of cores that may read from it. In addition there is also the data-value invariant that mandates that a read returns the value of the latest write to that location. Hieragen is currently restricted to protocols that enforce SWMR. Not all protocols, however, enforce SWMR. For instance, some recently-proposed protocols [14]–[17] relax SWMR and instead enforce the consistency model directly.

B. ProtoGen

ProtoGen [2] is a pillar on which we base HieraGen's automatic generation of concurrency. ProtoGen's input is a



Spec of SSP-H
atomic directory/ shared cache
atomic private cache
Spec of SSP-L
atomic directory/ shared cache
atomic private cache
Step #1: generate hierarchical SSP
atomic root
atomic cache-H
atomic dir/cache
atomic cache-L
Step #2: generate concurrent hierarchical protocol
concurrent root
concurrent cache-H
concurrent dir/cache
concurrent cache-L
Inputs: SSPs for each level [domain-specific language]
Intermediate Results: finite state machines for atomic hierarchical protocol [internal format for FSMs]
Outputs: finite state machines for concurrent hierarchical protocol [Murphi model checker language]

Fig. 2: HieraGen Tool Flow

SSP for a flat directory protocol, as written in a domain specific language. The input describes the behavior of a cache controller and the directory controller, assuming an atomic system model in which only one transaction is in-flight at a time. ProtoGen outputs the finite state machines of the cache controller and the directory. These two finite state machines are in the format of the Murϕ model checker [13], so as to facilitate verification.

ProtoGen overcomes the key challenge in generating (flat) protocols—creating cache and directory controllers that correctly handle incoming coherence messages when transactions are racing—by leveraging the insight that, in a directory-based coherence protocol, racing transactions are serialized at the directory. ProtoGen enables the directory to convey this serialization order to the caches via the forwarded requests it sends to caches; it overcomes possible ambiguities by renaming certain forwarded requests. With the caches and the directory achieving consensus on the order of racing transactions, ProtoGen can generate highly-concurrent and non-blocking (non-stalling) controller actions that are consistent with this order. ProtoGen also has a flag that limits concurrency by generating only stalling protocols. With the latter, cache and directory controllers stall when they receive potentially racing requests, at the cost of performance (while still preventing deadlocks). With the former, the generated protocol avoids stalling whenever possible at the expense of an increase in the number of transient states.

III. Baseline System Model and Terminology

For ease of explanation, we present a baseline system model with certain constraints that we will use when explaining HieraGen. In Section VII, we discuss the impact on HieraGen of relaxing some of these constraints.

Our baseline system model encompasses the two designs in Figure 1(b)-(c). For purposes of HieraGen, Figure 1(b) and Figure 1(c) are equivalent. Each directory/cache node tracks the coherence state of blocks held in the private caches of its children as well as blocks held by its collocated shared cache (if any). The root directory/cache is attached to main memory. These are both two-level designs, and we use "higher level" to refer to the level closer to the root and "lower level" to refer to the level farther from the root (e.g., the subtree on the bottom right of Figure 1(b)).

For brevity, we refer to the four distinct node types as: root (root directory/cache), cache-H (core/cache node in higher level protocol), cache-L (core/cache node in lower level protocol), and dir/cache (for the intermediate directory/cache node). We refer to the higher level and lower level SSPs as SSP-H and SSP-L, respectively.

We assume the following five constraints for now. In Section VII, we relax the first three of these constraints.

- There are only two levels of hierarchy.
- Directories are inclusive[2] and full-map, and evictions of read-only blocks are *not* silent, i.e., each directory has complete knowledge of its children's coherence state.
- Each SSP is a flat directory protocol.
- Shared caches are inclusive.
- All communication across hierarchy levels is strictly parent/child, i.e., a node cannot communicate directly with nodes in other levels. Communication within a hierarchy level is general.

Throughout this paper we use the terminology of Sorin et al. [1]. This terminology includes coherence states (e.g., transient state names like IS that denote the block is in a transient state between states I and S) and coherence requests (e.g., GetShared or GetS).

IV. HieraGen Tool Flow

HieraGen starts with an SSP for each level of the hierarchy, and it produces the finite state machines of all of the

[2]Directory inclusion means that a block may not be in a cache without the directory's knowledge.

distinct core/cache and directory/cache nodes. We illustrate HieraGen's flow from inputs to outputs in Figure 2. There are two main steps: (1) from flat SSPs to an atomic hierarchical protocol, and (2) from an atomic hierarchical protocol to a concurrent hierarchical protocol.

The SSPs are described in a domain-specific language (DSL); we employ the same DSL as that of ProtoGen. The final outputs are the finite state machines (FSMs) for concurrent hierarchical protocols. To enable verification, we produce the FSMs in the Murϕ model checker language. Next, we discuss the two steps in detail in the following sections.

V. Step 1: Atomic Hierarchical Protocol

All of the complexity in this step involves the generation of the dir/cache (intermediate directory/cache node). As highlighted in Figure 2, it is the only node that differs from the input specifications; the other nodes effectively pass through this step unchanged. Specifically, HieraGen must compose the cache-H from the higher level with the dir-L from the lower level into one intermediate dir/cache node. The dir/cache node must integrate the functionality of a directory to its children with the functionality of a child to its parent.

A. Intuition

HieraGen's philosophy is to perform the protocol composition in the most general way possible without being "aware" of specific protocols or states, i.e., we do not modify the input SSPs in any way. The key idea is to encapsulate the coherence actions of the other level within a coherence transaction, so that SWMR and data-value invariants are enforced globally. Specifically, before completing a read request originating from a particular level, HieraGen first ensures that the other level has no writers (if there is a writer, it is first downgraded). In a similar vein, before completing a write request, HieraGen ensures that the other level does not have any readers or writers (if there are any readers or writers, they are first invalidated).

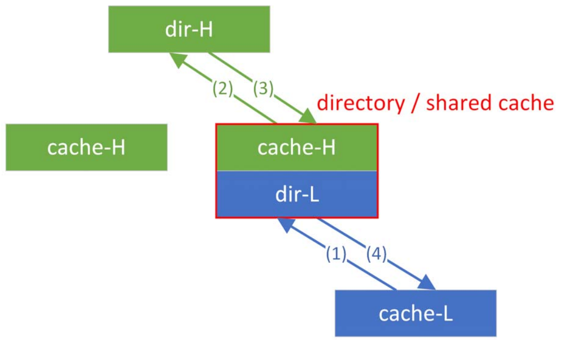


Fig. 3: Encapsulating higher-level coherence actions within a lower-level coherence transaction

How is this enforced by HieraGen? Consider Figure 3 which shows a coherence request originating from cache-L to dir-L (1). Suppose dir-L determines that the request cannot be completely satisfied at the lower level. HieraGen then encapsulates higher-level protocol actions within the lower-level transaction to ensure that the higher level is ready for the access to be performed in the lower level. By processing SSP-L, HieraGen is able to figure out the access type of the lower-level coherence request (whether it is a read or a write). By leveraging SSP-H, HieraGen makes cache-H generate a higher-level request of the same access type (2). Since SSP-H enforces SWMR in the higher level, when cache-H receives a response from dir-H (3), one can infer that the higher level is ready for the access to be performed in the lower level. Therefore, HieraGen resumes the lower-level coherence transaction, i.e., dir-L responds to cache-L (4), thus completing the lower-level coherence transaction while globally enforcing SWMR.

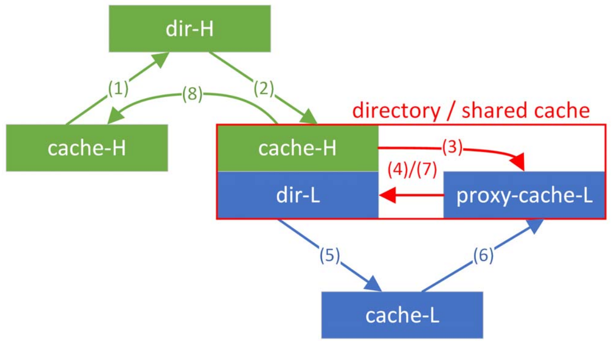


Fig. 4: Encapsulating lower-level coherence actions within a higher-level coherence transaction via the proxy-cache

Consider Figure 4 which shows a coherence request originating from one cache-H (1) and then forwarded by dir-H to another cache-H (2). Suppose the request cannot be compeletely satisfied at the higher level. HieraGen then encapsulates lower-level protocol actions within the higher-level transaction to ensure that the lower level is ready for the request to be completed in the higher level. Like in the previous scenario, HieraGen can leverage SSP-H to determine the access type of the higher-level request. But what entity can HieraGen leverage to make a lower-level request of that access type? Cache-H cannot be used because it logically belongs to the higher level. Dir-L cannot directly be used because read or write requests do not typically originate from the directory. Therefore, HieraGen clones a cache controller from SSP-L called *proxy-cache* and integrates it into the intermediate dir/cache node as shown in Figure 4. HieraGen makes the proxy-cache generate a coherence request to dir-L (4), which then forwards it to one or more caches in the lower level (5). When the proxy-cache receives a response (6), one can infer that the lower level is ready for the access to be performed in the higher level. The proxy-cache then proceeds to evict the block into cache-H (7), allowing cache-H to respond to the requestor (8), thereby enforcing SWMR globally.

B. HieraGen in Detail via Transaction Flow Examples

We now illustrate how HieraGen works by describing the protocol activity it must generate for every transaction using

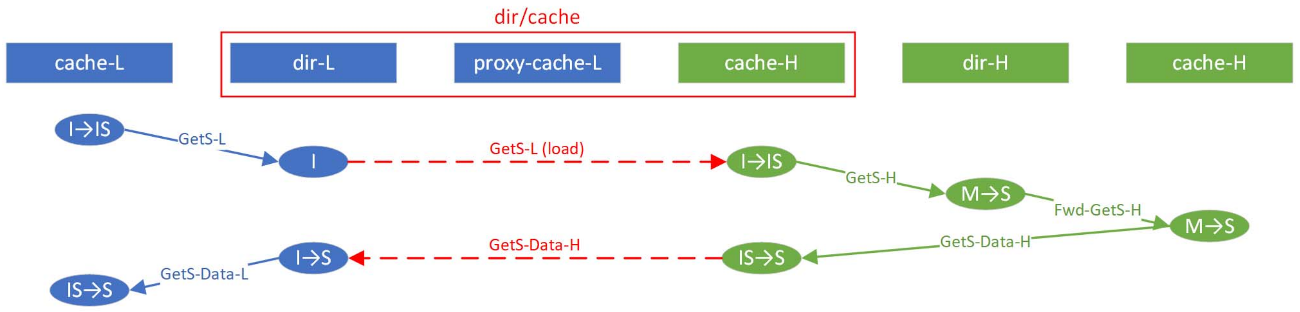


Fig. 5: Transaction Flow 1: A load from cache-L that involves the higher level. (Read from left to right and back). Dashed arrows denote messages internal to the dir/cache. (The proxy-cache-L is uninvolved in this transaction.)

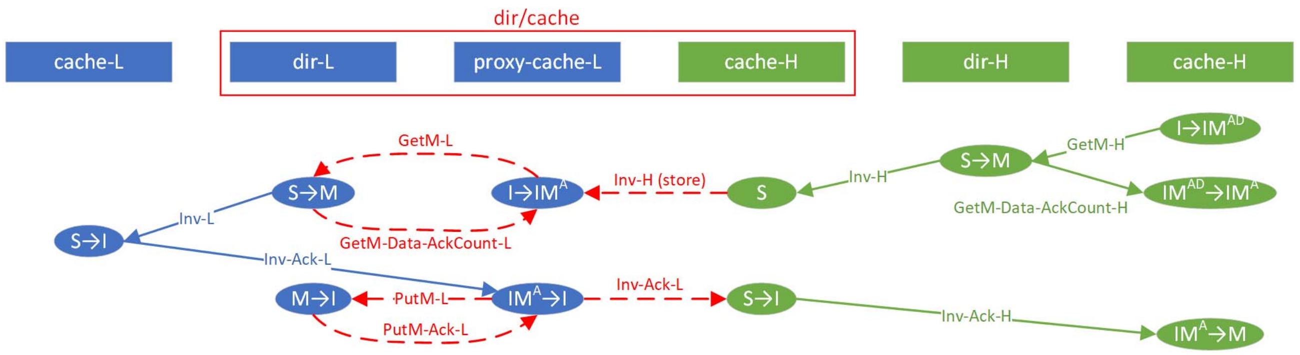


Fig. 6: Transaction Flow 2: A store from cache-H that involves the lower level. (Read from right to left and back). Dashed arrows denote messages internal to the dir/cache.

concrete examples. We assume for now that both levels of the hierarchy use typical MSI protocols.

Protocol transactions that do not require the dir/cache to provide both of its functionalities are effectively unchanged with respect to the input SSPs. A request from a cache-L that can be completely satisfied within SSP-L occurs as expected. For example, a GetShared (GetS-L[3]) request from a cache-L to the dir/cache that can be purely handled by the dir/cache—either by sending the data directly from the dir/cache or by forwarding the request to an owner that is a child of the dir/cache—behaves as in the original SSP-L. Similarly, a request from a cache-H that can be satisfied within SSP-H also occurs as expected; as long as the request does not require the involvement of dir-L from the dir/cache, it behaves as in the original SSP-H.

There are three types of transactions that require the dir/cache to provide both of its functionalities and thus require HieraGen to generate the corresponding dir/cache logic. We now walk through concrete examples of each type of transaction.

1) SSP-L loads/stores that involve SSP-H: Our first example is illustrated in Figure 5. Initially, a block is in state M in one cache-H, and the cache-L issues a GetS-L to the dir/cache. HieraGen can tell from processing SSP-L that this GetS-L request corresponds to a read, by inferring that the final state of the transaction allows only reads and not writes. Similarly, HieraGen can tell from processing SSP-H which coherence request a cache-H would issue if it needed to obtain (at least) read permissions, which is GetS-H. HieraGen matches these two request types; that is, because the cache-L's GetS-L provides read permissions, it has the dir/cache issue the GetS-H request to the dir-H (root) that a cache-H would issue for obtaining read permissions.

The dir-H and the rest of SSP-H behave as usual for that type of coherence request. In this example, the dir-H forwards the GetS-H for readable data to the cache-H that is the owner, and that owner responds with data to the dir/cache. The dir/cache fills its cache-H with the data and responds to the cache-L that made the original GetS-L request. This response is the same response as would be made to the original GetS-L request in a flat SSP-L.

2) SSP-H loads/stores that involve SSP-L: As illustrated in Figure 6, our concrete example here is a GetM-H request from a cache-H when one cache-L has the block in state S (and no other caches have the block). The cache-H sends a GetM-H to the dir-H, and the dir-H does two things: it forwards the request (in the form of an Invalidation-H) to the dir/cache, and it sends a message to the requesting cache-H to let it know how many Acknowledgments to expect. HieraGen can tell from processing SSP-H that the Invalidation-H corresponds to a write (and not a read), and it needs the dir/cache to emulate what should happen due to a request for writable

[3]When otherwise ambiguous, we label coherence messages with "-L" or "-H" to denote which protocol they are in.

data. HieraGen provides this functionality via the proxy-cache introduced in the previous section. (Recall that the proxy-cache is essentially a clone of the cache-L controller that is integrated as part of the dir/cache; it is used to encapsulate coherence transactions, but it does not perform loads or stores.)

HieraGen can tell from processing SSP-L which coherence request a cache-L would issue if it needed to obtain write permissions, which is GetM-L, and it has the proxy-cache-L issue a GetM-L to dir-L. This GetM-L is an internal request since the proxy-cache-L and dir-L are part of the same controller. The dir-L then sends an Invalidation-L to the cache-L in state S and transitions to state M (because it now views the proxy-cache-L as being in state M). This GetM-L transaction completes once the cache-L in state S has sent an Invalidation-Ack-L to the proxy-cache-L; the transaction ensures that all of the cache-L nodes are in the appropriate coherence state (Invalid) with respect to the cache-H that made the original GetM-H request.

Once the GetM-L transaction completes, the proxy-cache-L immediately evicts the cache block into the dir/cache, causing dir-L's state to change from M to I. The dir/cache then responds to the root with the appropriate response for SSP-H (which is an Invalidation-Ack-H in the example). Note that the response would have included the data if one of the cache-L nodes had been the owner.

3) Dir/cache evictions: To maintain directory inclusion, an eviction from the dir/cache must first evict the block from all cache-L nodes, if any, that have the block. HieraGen again employs the proxy-cache-L for this purpose, and HieraGen exploits its ability to process the SSP-L to discover which SSP-L coherence request invalidates the block from all cache-L nodes. Thus, the proxy-cache-L issues a GetM-L, resulting in the proxy-cache-L holding the only copy of the block in SSP-L. (If a cache-L is the owner when it is invalidated, it sends its data to the proxy-cache-L.) The proxy-cache-L then evicts the block to the dir/cache. Once the only copy is at the dir/cache, the dir/cache issues a PutM-H to the root, the coherence request for evicting an owned block in SSP-H.

C. Algorithm

We now precisely illustrate how our algorithm composes cache-H, dir-L, and cache-L (proxy cache) to produce the intermediate dir/cache controller. Our algorithm essentially takes "code" from the input controllers and stitches them together. Therefore, to specify how our algorithm works, we must first provide a notation for the controllers.

In the following discussion, "Accesses" refers to the set of accesses: load, store, and evict. "States", "Requests", and "Fwd-requests" refer to the sets of states, requests, and forwarded requests (resp.) associated with a controller. For example, cache-L.States, refers to the set of stable states associated with the lower level cache controller. Similarly, "send-request", "send-fwd-request", "await-response", "update-state", and "send-response" are variables that point to code that do what their names indicate.

1) Cache controller: The cache controller component of an SSP specifies, for each stable state, what happens on each access and each incoming forwarded request. We specify this as follows. (Note that this abstract specification must be instantiated to make up specific cache controllers as the following example illustrates.)

```
∀ access ∈ Accesses, ∀ state ∈ cache.States
    cache.send-request(access,state);
    cache.await-response(access,state);
    cache.update-state(access,state);

∀ fwd-request ∈ cache.Fwd-requests, ∀ state ∈ cache.States
    cache.update-state(fwd-request,state);
    cache.send-response(fwd-request,state);
```

Consider a lower level MSI cache controller: cache-L.send-request(store,I) points to code that sends a GetM to dir-L; cache-L.await-response(store,I) points to code that waits for Data; and cache.update-state(store,I) points to code that changes state from I to M. Some of these code pointers may point to empty actions—for example, a store to a block in state M needs no messages to be sent nor any state update.

2) Directory controller: The directory controller component of an SSP specifies, for each stable state, what happens on an incoming request. We specify this as follows. (As before, the abstract specification has to be instantiated to make up specific directory controllers.)

```
∀ request ∈ dir.Requests, ∀ state ∈ dir.States
    dir.send-fwd-request(request,state);
    dir.await-response(request,state);
    dir.update-state(request,state);
```

3) Generating dir/cache controller: We can now specify how dir-L, cache-H, and cache-L are composed to form the intermediate dir/cache controller. This compound controller, being a directory as well as a cache, will have to specify for the cross-product of dir-L/cache-H states, what happens on: (a) an incoming request from a cache-L (Figure 3); and (b) an incoming forwarded request from dir-H (Figure 4). We consider the former first, as shown below.

```
∀ request ∈ dir-L.Requests
∀ (dir-state, cache-state) ∈ dir-L.States × cache-H.States

 /* compute access that generated request at lower level */
   access = compute_access(request, SSP-L);

 /* issue request to the higher level of same access type */
   cache-H.send-request(access, cache-state);
   cache-H.await-response(access, cache-state);
   cache-H.update-state(access, cache-state);

 /* Now respond to the request in the lower level */
   dir-L.send-fwd-request(request, dir-state);
   dir-L.await-response(request, dir-state);
   dir-L.update-state(request, dir-state);
```

By parsing SSP-L, we determine the access that leads to the request. We then make the controller send a request to the higher level of the same access type (if necessary). Finally, we have the controller respond to the original request.

```
∀ fwd-request ∈ cache-H.Fwd-requests
∀ (dir-state, cache-state) ∈ dir-L.States × cache-H.States

 /* compute access that generated fwd-request at higher level */
    access = compute_access(fwd-request, SSP-H);

 /* The proxy cache logically issues a request of the same access
 to dir-L. But this request, being internal, needn't actually be sent.
 We simply compute the request it would generate (from Invalid
 state) so that dir-L can respond to this virtual request*/

    request = compute_request(access, Invalid, SSP-L);
    dir-L.send-fwd-request(request, dir-state);
    dir-L.await-response(request, dir-state);
    dir-L.update-state(request, dir-state);

 /* The virtual proxy cache waits for response */
    cache-L.await-response(access, Invalid);
 /* Then the proxy cache updates its state */
    final-state = cache-L.update-state(access, Invalid);

 /* The proxy cache must now evict the block, but this is again
 an internal request and needn't be sent. We simply compute the
 request it would generate so that dir-L can respond to this virtual
 eviction request */

   evict-request = compute_request(evict, final-state, SSP-L);
    dir-L.update-state(evict-request, dir-state);

 /* Now respond to the forwarded request */
    cache-H.update-state(fwd-request, cache-state);
    cache-H.send-response(fwd-request, cache-state);
```

Next, we deal with incoming forwarded requests. As shown above, the idea is to first compute the access that led to the forwarded request by parsing SSP-H. (It is worth reiterating that this computation, and indeed all of the following steps, happen at design time.) Then, we must make the controller "perform" the access in the lower level. The "code" for how to do this is available in the input SSP-L, and the source of this transaction is the cache-L. That is why we leverage a proxy cache to initiate this transaction. In reality, the proxy cache is a single temporary cache line (and state) which is physically integrated within the dir/cache controller. Therefore the request from the proxy cache to dir-L need not be physically sent. Instead, we compute (at design time) what request it would have sent and simply make dir-L react to this request. We then make the controller await the response that the proxy cache would have normally waited for. Once the response is received, we make the controller update its state and make it evict the block into dir-L. Again this eviction is virtual and so no message is actually sent. Instead the request corresponding to the evict access is computed, and dir-L is made to react to this request. Finally, cache-H is made to respond to the incoming forwarded request from the higher level.

D. Compatibility Between Protocol Levels

HieraGen can compose SSPs together into a hierarchical protocol, but not all SSPs are immediatly compatible. Specifically, there is one protocol feature—silent upgrading of coherence permissions—that can cause incompatibility if not handled appropriately. In "typical" protocols that use a subset of the MOESI stable coherence states, the culprit is the (E)xclusive state. In the E state, which is read-only, a cache can silently upgrade to the M state, which is read-write.

The issue of protocol compatibility is best explained through an example of incompatibility. Consider the case in which SSP-L is MESI and SSP-H is MSI. Assume initially that the block is Invalid in all caches. One cache-L performs a load, misses, and issues a GetS-L to the dir/cache. The dir/cache issues a GetS-H to the root, and the root responds to the dir/cache with Shared (read-only) permission and data. The root records that its dir/cache child is in state S. Because SSP-L is MESI and there were no sharers at the time of the GetS-L from the requesting cache-L, the dir/cache responds to the cache-L with data and Exclusive permissions. Now the cache-L can silently transition from E to M and write the block. Meanwhile, any cache-H can issue a GetS-H to the root and obtain Shared access. In this situation, the hierarchical protocol violates the SWMR coherence invariant.

Fortunately, HieraGen can automatically detect incompatibility when processing the SSPs, because it can detect when an SSP permits silent permission upgrades. In our example above, since the cache-L can silently transition from E to M and write the block, HieraGen can infer that E state is writable. There are two solutions to this problem. The first one is more intuitive, but the second one offers better performance.

The intuitive solution is for HieraGen to have the dir/cache conservatively issue a GetM-H request that corresponds to the greatest permissions that the cache-L could receive (in this case, read-write access), rather than the GetS-H request that corresponds to the original request (for read-only access). While this solution ensures safety, it has negative performance implications due to needless SSP-H invalidations if the originating cache-L does not write to the block.

The higher performance solution avoids these needless invalidations. The dir/cache issues a GetS-H, and the root responds to the dir/cache with read-only access. HieraGen, when generating the dir/cache, adds logic to detect mismatches between the permission its cache-H received from SSP-H (S=read-only) and the permission its dir-L would otherwise grant to the cache-L requestor (E=silently upgradeable to read-write). In this case, it has the proxy-cache-L issue a request for the *received* permission, i.e., the proxy-cache-L issues a GetS-L. In this way, the proxy-cache-L mimics the possible behavior of an external cache-H. The proxy-cache-L's

GetS-L, which is serialized before the cache-L's GetS-L, puts the dir-L momentarily in state E, from which it will now grant S (not E) permissions to the requesting cache-L. Once the dir/cache has responded to the cache-L, the dir-L changes to state S, and the proxy-cache-L evicts its block.

With this more optimized solution, there is one last issue to resolve. Let us assume that both the SSP-H and SSP-L have protocols with E states. Initially let us assume that all blocks are in I state. Consider the situation in which cache-L issues a GetS-L to request a block for reading; the dir/cache first obtains the block in state E (due to SSP-H) and provides the block in E state for the requesting cache-L (due to SSP-L). Now the cache-L can silently upgrade to M and modify the block, without notifying the dir/cache, as per SSP-L. We now have another mismatch to resolve. We want the dir/cache's cache-H to change to state M so that it is compatible with the earlier store that was performed at cache-L. Furthermore, we want to do this without modifying SSP-H, which we recall is HieraGen's philosophy.

When cache-L evicts the block to the dir/cache, the type of eviction message (PutE-L or PutM-L) reveals the access that was performed in the cache-L. In our case, PutM-L reveals that a write occurred. Therefore, there must now be a write of cache-H to update it with the evicted data. But because cache-H is already in state E, it does not need to issue a request to dir-H; it can silently go to state M as per SSP-H.

E. *Optimizing Protocol Finite State Machines*

As explained thus far, HieraGen will create somewhat unoptimized finite state machines for the root and dir/cache. Consider a state machine to be a 2D matrix, in which rows are coherence states, columns are events (incoming coherence messages), and entries specify what happens for that state/event pair. Naively, a specification of a directory (either the root or the directory part of the dir/cache) or a cache controller would have an entry for every possible state/event pair, i.e., NumRows*NumCols entries. However, many of those entries are not actually reachable, because certain events cannot occur in certain states.

Without optimization, the finite state machines for directories will needlessly include logic to handle unreachable state/event pairs. This logic is not harmful, but it is unnecessary; it may also complicate debugging in that it may be useful to know that a believed-to-be unreachable state/event has been reached. We have implemented a custom model checker that explores the reachable state space of the protocols so as to eliminate these unreachable state/event pairs.

VI. Step 2: Concurrent Hierarchical Protocol

At the end of Step 1, HieraGen has produced an atomic hierarchical protocol in the form of finite state machines for the cache-L, dir/cache, cache-H, and root. In Step 2, we add concurrency to this protocol.

For flat protocols, ProtoGen injected concurrency as explained in Section II-B. ProtoGen could leverage the fact that every coherence transaction was serialized at the directory.

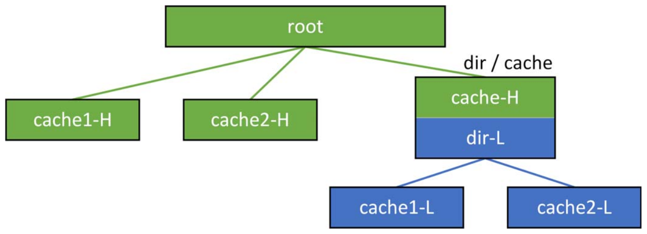


Fig. 7: There is a unique serialization point for any two racing transactions.

(There is just one directory in a flat protocol). HieraGen can still leverage transaction serialization, but in a hierarchical protocol, there can be multiple serialization points, depending on the protocol level of the block's current owner. Consider the system model in Figure 7. A request from a cache-L that is satisfied entirely within SSP-L is serialized at the dir/cache. However, a request from a cache-H or a request from a cache-L that cannot be satisfied within SSP-L is serialized at the root.

Despite the two serialization points (or more, if more levels of hierarchy), the tree structure of the hierarchy provides the invariant that any two racing coherence transactions are serialized at exactly one of these serialization points. ProtoGen can thus be leveraged for extracting concurrency.

Consider an example in which cache1-H holds a block in writable state. Assume two racing transactions: cache2-H and cache1-L both want to write to the block. Accordingly, both of their requests will attempt to obtain ownership of the block; the request that reaches the root first will win the race. In other words, the root is the serialization point.

For the same initial state—i.e., cache1-H initially holding the block in writable state—let us now consider two racing transactions coming from the lower level. Specifically, assume both cache1-L and cache2-L want to obtain ownership of the block. In this situation, although there are potentially two serialization points in play, the hierarchical nature means that the first transaction to reach the dir/cache wins the race; the transaction to reach the dir/cache second will be able to infer that it has lost the race and will not proceed to the root. In other words, the dir/cache is the serialization point.

Let us now generalize. Because a HieraGen-generated atomic protocol enforces SWMR globally, there can be exactly one owner for any block. (It is the cache that holds the block in writable state; if there is no such cache, the owner is the root). Any two racing transactions, therefore, will both attempt to reach this unique owner by traversing a path consisting of one or more directory nodes. The tree structure guarantees there will be exactly one directory node in common across the two paths. This is because the transaction that reaches this common node second can infer that it has lost and will not proceed any further towards the original owner. In other words, this common directory will serve as the unique serialization point, allowing us to use ProtoGen for performing Step 2.

VII. Other System Models

In Section III, we presented our baseline system model, and we listed five design constraints that we imposed at the time.

We now explore the effects of relaxing each of them.

A. Deeper Hierarchies

As systems continue to scale, there is likely to be incentive to use more levels of hierarchy. We consider whether deeper hierarchies affect how HieraGen composes SSPs into a hierarchy (Step 1) and how HieraGen introduces concurrency (Step 2).

1) Step-1: Composition is unaffected by the depth of the hierarchy, for two reasons. First, at each point of SSP composition, there is a structured interface consisting of a single dir/cache node that provides the cache functionality to its parent and the directory functionality to its children. This structured interface is used by HieraGen to ensure that before a coherence request from one level completes, each of the other levels are in a state that allows for this coherence request to complete without violating SWMR globally. Second, HieraGen does not permit any communication across levels, without this structured interface. Thus, the reasons for why HieraGen works for composing two SSPs into a hierarchy also apply for composing additional SSPs.

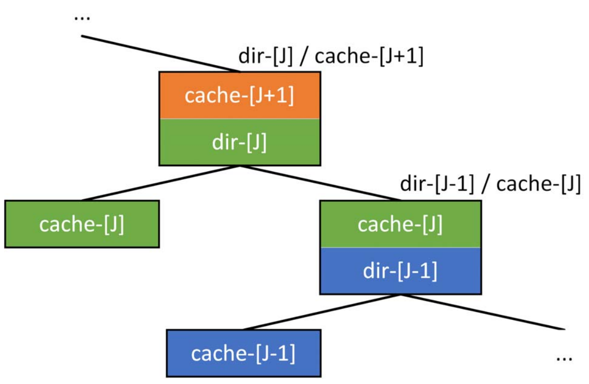


Fig. 8: How HieraGen works with deeper hierarchies. A write from cache-J leads to a write in the higher level (via dir–[J]/cache–[J+1]) and write in the lower level (via dir–[j-1]/cache–[J])

Figure 8 illustrates how HieraGen's tree structure ensures SWMR globally. Consider an n-level hierarchy in which cache-J (a cache from the j^{th}) level performs a write. Assume that there are one or more sharers in level(s) higher than j as well as level(s) lower. Before cache-J's write request is completed, HieraGen issues write requests to the higher levels (via dir–J/cache–[J+1]) as well as lower levels (via dir–[j-1]/cache–J), thereby ensuring SWMR globally.

2) Step-2: Similarly, deeper hierarchies have no impact on how HieraGen introduces concurrency into the hierarchical protocol. This is because, irrespective of the depth of the hierarchy, any two racing transactions serialize at exactly one node. This enables the use of ProtoGen to uncover concurrency.

B. Incomplete Directory Knowledge

There are three ways in which a directory can have incomplete or stale knowledge of its children's coherence states: the directory uses an incomplete data structure (e.g., coarse sharing vector), its child caches are permitted to perform silent evictions of read-only blocks, or the directory is not inclusive. Fortunately, this design issue does not affect HieraGen because it is handled in the input SSPs.

For example, assume that SSP-L uses a non-inclusive directory. Recall that in a non-inclusive directory, a directory miss does not mean that the block is uncached in any of its children. Therefore, to ensure SWMR, SSP-L would already have had to revert to a broadcast on a write to a block missing in the directory. This feature of SSP-L is what HieraGen leverages to ensure SWMR at the level, and hence globally.

C. Other SSP Protocol Types

We have assumed that each SSP is a flat directory protocol, but there is some flexibility here. The key is that the protocol must have a single structure that can serve as an interface between hierarchy levels. Directory protocols naturally have that structure: the directory.

However, a snooping protocol can also be viewed as a directory protocol with a "null directory," sometimes denoted Dir_1B. In such a snooping protocol, every coherence request is sent to the "directory," and the stateless directory simply broadcasts the request to all of its children. As long as the interconnection network provides point-to-point ordering between the "directory" and each of its children, this protocol will provide broadcast snooping.

D. Non-Inclusive Shared Caches

We have thus far assumed that shared caches are inclusive, but there are systems that provide either exclusion or non-inclusion (i.e., neither strictly inclusive nor exclusive).

In its current form, HieraGen is limited to inclusive shared caches. The underlying reason for this limitation is our philosophy of (a) allowing the user to specify the SSPs completely independently, and (b) not modifying the SSPs when composing them. Because we have drawn a sharp line between the SSPs, they have no knowledge of each other.

To illustrate the problem with non-inclusion, consider the following scenario. One cache-L is in M, its dir-L knows that the cache-L is in M, and the block is in M in the shared cache (i.e., the cache part of the dir/cache). In a non-inclusive model, the shared cache would consult with the dir-L to decide how to proceed. If any of its cache-L children still has the block in state M, the shared cache can evict silently; else, the shared cache has the only up-to-date owned copy, and it must not drop it silently. However, with our separation between SSPs, the shared cache cannot consult the dir-L in this fashion, and thus it cannot know how to proceed.

Future work will explore the possibility of relaxing the sharp break between the SSPs, in order to enable non-inclusive shared caches.

E. Communication Across Levels

We have assumed a tree-structured hierarchy in which communication is strictly hierarchical. A node can communicate only with its parent, children, or siblings. Thus, for example, a cache-L cannot directly communicate with the root or a cache-H. This assumption enables us to clearly reason about the composition of SSPs, and it is critical to the current design and implementation of HieraGen. We can imagine a future tool that overcomes this limitation, but we leave this project to future work.

VIII. Experimental Evaluation

The goal of this evaluation is to determine the effectiveness of HieraGen in producing concurrent, hierarchical protocols. To illustrate the design automation benefits of HieraGen, we first compare the complexity of the input SSPs to the complexity of the hierarchical SSP and the concurrent hierarchical protocol. We then discuss the verification of the generated protocols.

We view hierarchy as useful—as do the architects of existing hierarchical protocols—and we do not perform experiments to quantitatively confirm its benefits.

A. Benchmarks

Our "benchmarks" are flat input SSPs along with the descriptions of how the hierarchy is structured (e.g., that SSP-L is attached to SSP-H at a specified point). These SSPs include typical MSI, MESI, MOSI, and MOESI protocols, like those found in Sorin et al. [1] but without the concurrency.

In Table I, we present the complexity of the flat input SSPs. Complexity is difficult to quantify precisely but, for a $flat$ coherence protocol, the numbers of states and reachable state/event pairs (i.e., transitions) are reasonable proxies.

Protocol	Cache	Directory
MI	2/9	2/6
MSI	3/26	3/16
MESI	4/33	4/25
MOSI	4/38	4/24
MOESI	5/45	5/33

TABLE I: Flat atomic protocols. Each entry is the number of stable states/transitions.

SSP-L/SSP-H	dir-L	cache-H	dir/cache
MSI/MI	4/16	5/9	10/42
MI/MSI	2/4	10/26	12/37
MSI/MSI	4/16	10/26	21/94
MESI/MSI	6/25	10/26	26/119
MESI/MESI	6/25	12/33	40/184
MOSI/MSI	4/24	10/26	28/149
MOSI/MOSI	4/24	14/38	42/227
MOESI/MOESI	5/33	16/45	59/368

TABLE II: Complexity of atomic hierarchical protocols produced by HieraGen. Each entry is the number of states(stable+transient)/transitions.

B. Design Complexity

The goal of HieraGen is to overcome the design complexity of manually designing hierarchical protocols. Thus we provide HieraGen with pairs of input SSPs, one SSP-L and one SSP-H, and study the concurrent hierarchical protocols it produces.

For additional insight, we first study the results of Step 1, i.e., the atomic hierarchical protocols. In Table II, we show the quantifiable complexity results for seven different hierarchical protocols. Each row of the table compares, for a given hierarchical protocol, the complexity of the dir-L and cache-H of the input SSPs to the automatically-generated dir/cache. We note first that the dir-L and cache-H appear to have greater complexity than they did in Table I; this discrepancy is because we are now considering the dir-L and cache-H after HieraGen has expanded them to include transient states (but still no concurrency).[4] Potential races introduced due to these transient states will have to be considered in Step 2, and hence we report these transient states.

Looking at these results, we observe that the complexity introduced in Step 1 can be considerable, and it varies considerably across the protocols. At the low end, the dir/cache in the MSI/MI hierarchy, when compared to the sum of its constituent parts, has only one more state and roughly double the number of reachable transitions. At the high end, the MOESI/MOESI dir/cache has 59 states and 368 transitions, far more than the sum of its parts (21 states and 78 transitions). We also remind the reader that not all states and transitions are equally easy to reason about, and we find it particularly challenging to reason about the dir/cache and how it must bridge the two SSPs.

We now examine the additional complexity introduced during Step 2, when HieraGen adds concurrency to the protocols. We run HieraGen twice for each protocol, once with the input flag set to produce a stalling protocol and once with the flag unset. Table III shows the quantifiable complexity for HieraGen's final output; the table reproduces the results from Table II to facilitate visual comparisons.

One perhaps curious result is that the nodes in the concurrent protocols often have approximately the same—or even fewer—states than the corresponding atomic protocols. This phenomena is because, even though the protocol complexity has increased, HieraGen can often discover how to merge states that are equivalent. For example, a cache in an MSI protocol may have states MI and SI, which denote that the cache has evicted a block in state M or state S, respectively. Because all messages that arrive in these states are distinct, they can be merged. Manual protocol designers, including the authors, tend not to merge states in this way. In fact, for clarity when reading and debugging a protocol, designers are likely to want to distinguish states like these, but HieraGen does not need to care as much about readability or debuggability (since the generated protocols are correct by construction).

Ultimately, what is more important than the quantifiable complexity metrics is HieraGen's ability to automatically and nearly instantaneously produce protocols that are correct by construction. HieraGen took less than 10 seconds to correctly generate each of the protocols in this section.

[4]Even atomic protocols have transient states. Atomicity simply means that messages from other transactions cannot intervene in transient states.

	Atomic Hierarchical				Concurrent Hierarchical Stalling				Concurrent Hierarchical Non-stalling			
SSP-L/SSP-H	cache-L	dir/cache	cache-H	root	cache-L	dir/cache	cache-H	root	cache-L	dir/cache	cache-H	root
MSI/MI	10/26	10/42	5/9	2/4	9/31	10/43	4/9	2/6	12/42	10/43	4/9	2/6
MI/MSI	5/9	12/37	10/26	4/16	4/9	12/41	9/31	4/24	4/9	16/53	12/42	4/24
MSI/MSI	10/26	21/94	10/26	4/16	9/31	21/102	9/31	4/24	12/42	28/126	12/42	4/24
MESI/MSI	12/33	26/119	10/26	4/16	10/37	26/126	9/31	4/24	13/48	31/145	12/42	4/24
MESI/MESI	12/33	40/184	12/33	6/25	10/37	39/191	10/37	6/45	13/48	44/210	13/48	6/45
MOSI/MSI	14/38	28/149	10/26	4/16	10/40	31/170	9/31	4/24	13/51	39/206	12/42	4/24
MOSI/MOSI	14/38	42/227	14/38	4/24	10/40	47/273	10/40	4/37	13/51	64/353	13/51	4/37
MOESI/MOESI	16/45	59/368	16/45	5/33	11/46	64/415	11/46	5/66	14/57	81/495	14/57	5/66

TABLE III: Concurrent hierarchical protocols. This table shows the complexities of the HieraGen-generated concurrent cache-L, dir/cache, cache-H, and root nodes compared with their atomic counterparts. We present both stalling and non-stalling protocol variants. Each entry is the number of states(stable+transient)/transitions.

C. Verification of Correctness

To confirm HieraGen produces correct protocols—protocols that never violate coherence and never deadlock—we perform three verifications for every protocol.

First, we use the Murϕ model checker [13] to formally and completely verify the atomic hierarchical protocol that is produced by Step 1 of HieraGen for a configuration depicted in Figure 1b and Figure 1d. The configuration consists of a single root directory, two cache-H nodes, the generated dir/cache (including the proxy-cache-L), and two cache-L nodes.

Second, we verify the concurrent hierarchical protocols generated by HieraGen for the same configuration. The verification is performed on a server with 256GB of memory.

Third, to gain more confidence we add one additional cache-L node, resulting in a configuration consisting of: a single root directory, two cache-H nodes, the generated dir/cache (including the proxy-cache-L), and three cache-L nodes. However, Murϕ runs out of memory for this configuration. To extend the verification to this configuration, we used the hash compaction capability provided by Murϕ [18]. Hash compaction compresses the state descriptors stored within the state table to reduce the memory footprint of the model checker during verification. Due to the compression, there exists a small but non-zero probability that system states are omitted during verification; after each verification run, Murϕ model reports this omission probability. Because Murϕ randomly picks independent compression functions for the state descriptors, the omission probabilities of different runs can be multiplied. For each coherence protocol, we performed multiple verification runs, until the probability of an undetected bug fell below a threshold of 0.001%.

IX. Related Work

There are three primary areas of related work: frameworks for structured hierarchical protocols, design automation for coherence protocols, and hierarchical protocols that are designed for verifiability.

MCP [3] is a design framework that seeks to minimize design complexity by cleanly separating the two functionalities of the dir/cache (our term) into the manager (directory) and client (cache). Unlike HieraGen, MCP does not provide any design automation. Cook [19] automates by providing a protocol communication template. The template has many nice features, including hierarchy, but several critical constraints, including: blocking directories, no sibling-sibling communication, and caches that block during writebacks.

The second area of related work is in design automation for coherence protocols. We have already discussed ProtoGen [2] at length, but there were schemes prior to it. Dave et al. [20] used the Bluespec hardware description language to describe concurrent protocols, and the compiler produced RTL. Unlike HieraGen, this work starts with a concurrent protocol and its contribution is that it allows designers to write in Bluespec [21] instead of directly in RTL. One can also use Bluespec to specify an atomic protocol and have the compiler generate concurrency; however, the concurrency is quite limited because the compiler must conservatively serialize coherence transactions to the same cache block. Work by Staunstrup and Greenstreet [22] is similar but uses a different language called Synchronized Transitions. One other approach is to use ideas from program synthesis, i.e., specify part of the protocol and synthesize the rest. Examples include TRANSIT [23] and VerC3 [24], both of which are highly constrained by the state space explosion problem.

The third area of related work is in hierarchical protocol design that facilitates verification. Verifiability can ensure that the bugs that are likely to be introduced during manual protocol design are caught, but verifiability does not necessarily simplify the design process. Verifiable hierarchical protocols include HCC [6], Fractal Coherence [10], and protocols that conform to the Neo framework Neo [9].

X. Conclusions

We have presented HieraGen, a new tool for automatic generation of hierarchical cache coherence protocols. We have demonstrated that HieraGen can successfully compose atomic, flat SSPs into a highly concurrent hierarchical protocol. The generated protocols are verifiably correct and avoid the substantial design and verification effort—cache and directory controllers with dozens of states and hundreds of transitions—required by manual design. We believe that design automation of hierarchical protocols is both practical and preferable to manual design.

XI. Acknowledgments

We thank Atefeh Mehrabi, Theo Olausson, and the anonymous reviewers for their helpful feedback on this paper.

References

[1] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[2] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018, pp. 247–260.

[3] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-client pairing: A framework for implementing coherence hierarchies," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 226–236.

[4] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren, "Architecture and design of AlphaServer GS320," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 13–24.

[5] E. Hagersten and M. Koster, "Wildfire: A scalable path for SMPs," in *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, 1999, pp. 172–181.

[6] E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 11–22.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[8] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 46–56.

[9] O. Matthews and D. J. Sorin, "Architecting hierarchical coherence protocols for push-button parametric verification," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 477–489.

[10] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO*, 2010, pp. 471–482.

[11] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[12] O. Matthews, J. Bingham, and D. J. Sorin, "Verifiable hierarchical protocols with network invariants on parametric systems," in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, 2016, pp. 101–108.

[13] D. L. Dill, "The Murphi Verification System," in *Proceedings of the 8th International Conference on Computer Aided Verification*, 1996, pp. 390–393.

[14] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory coherence in the age of multicores," in *ICCD*, 2011.

[15] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, 2014.

[16] A. Ros and S. Kaxiras, "Racer: TSO consistency via race detection," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2016, pp. 33:1–33:13.

[17] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for GPUs," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2016, pp. 26:1–26:13.

[18] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995, pp. 206–224.

[19] H. Cook, "Productive design of extensible on-chip memory hierarchies," Ph.D. dissertation, University of California, Berkeley, 2016.

[20] N. Dave, M. C. Ng, and Arvind, "Automatic synthesis of cache-coherence protocol processors using bluespec," in *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2005, pp. 25–34.

[21] "Bluespec system verilog," http://bluespec.com/, note = Accessed: 2018-03-30.

[22] J. Staunstrup and M. R. Greenstreet, "From high-level descriptions to VLSI circuits," *BIT*, vol. 28, no. 3, pp. 620–638, 1988.

[23] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," 2013.

[24] M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan, "VerC3: A library for explicit state synthesis of concurrent systems," in *DATE*, 2018.