Specification and Verification of LambdaRAM– A Wide-area Distributed Cache for High Performance Computing

Venkatram Vishwanath, Lenore D. Zuck, Jason Leigh
Dept. of Computer Science,
University of Illinois at Chicago,
venkat@evl.uic.edu, lenore@cs.uic.edu, spiff@uic.edu

Abstract

LambdaRAM *is a high-performance, multi-dimensional, wide-area, distributed cache that takes advantage of massively available memory from multiple clusters interconnected by ultra high-speed networking to provide data-intensive scientific applications with rapid access to both local and remote data without suffering the latency bottlenecks often associated with large storage systems and wide-area data access.* LambdaRAM *has been demonstrated to yield significant performance speed-ups for Geophysical and Bioscience applications accessing extremely large datasets. Currently,* LambdaRAM *is being integrated by NASA for the Modelling, Analysis and Prediction (MAP) Program applications to study tropical cyclones. Formal verification of* LambdaRAM *is important to NASA to ensure that* LambdaRAM *operates reliably in real-time mission critical deployments.*

We present our preliminary steps towards full formal verification of LambdaRAM. *We first give an abstract description of the system and then verify several of its properties. Most of the proofs are accomplished by automatic techniques, while some require deductive steps.*

1 Introduction

Interactive real time exploration and correlation of multi-terabyte and petabyte datasets from multiple sources has been identified as a critical enabler for scientists to glean new insights in a variety of disciplines critical for national security. These include, e.g., climate modeling and prediction, biomedical imaging, geosciences, and high-energy physics [11]. The critical performance bottlenecks in such data-intensive applications are the access latencies associated with storage systems and remote data access. In NASA's Climate Modeling, Analysis, and Prediction (*MAP*) project [8], the access latencies cause the computational clusters, used in the GEOS5 global forecast model calculation [7], to idle for 25-50% of the execution time during the analysis phase. Reducing I/O latency while enabling real-time, high-performance data access and data sharing would allow employing more complex models, which, in turn, will result in faster and more accurate weather prediction and forecasts. Formal verification of these high-performance computing (HPC) systems is considered to be of utmost importance because they are often deployed in safety-critical environments. Yet, the vast number of parameters they depend on, the variable message latency and routing paths, as well as many other features, render their verification extremely challenging.

The OptIPuter [16] is a new paradigm in data-intensive distributed HPC whose goal is to build planetary-scale supercomputer that enables real-time, data-intensive HPC. This is achieved by interconnecting distributed storage, computing, and visualization resources over ultra-high speed photonic networks at tens of gigabits per second. LambdaRAM [17], the memory-subsystem of the OptIPuter, is a high-performance, multi-dimensional, distributed RAM-disk abstraction that takes advantage of massively available memory from multiple clusters and ultra high-speed networking, to provide data-intensive scientific applications with rapid access to both local and remote data. LambdaRAM employs latency mitigation heuristics, including pre-fetching, pre-sending, and hybrid heuristics, based on the access patterns of an application, and, novel data transport protocols designed for ultra high-badwidth networks, to mitigate data access latencies. It enables time-critical data collaboration between applications over geographically distributed clusters by providing a shared cache over the clusters interconnected by optical networks. Geophysical and Bioscience scientific data-analysis applications, using LambdaRAM to access remote datasets, have demonstrated large speedups [9].

NASA is currently integrating LambdaRAM with the MAP project for mission-critical deployment in tropical hurricane analysis. Consequently, the formal verification

of the LambdaRAM protocol has become a high priority. However, the only description of LambdaRAM is its over 30,000 C++ lines, which deems its formal verification virtually impossible.

In this paper, we report on our initial successful experience at both abstracting and formally verifying (the abstraction of) the LambdaRAM code. We believe that the work reported here is one of the first formal verifications of a protocol for high-performance computing. Moreover, in addition to describing the application of various theoretical techniques, it provides an encouraging evidence to the benefits of collaboration between system developers and formal method researchers. In particular:

- The formal verification of the LambdaRAM protocol was initiated to aid in the identification and elimination of bugs that were causing erroneous behavior as LambdaRAM was scaled to large cluster systems, and to enable its deployment in mission critical applications that require some certification. Although, the development of the code followed a pattern which violates widely accepted software engineering theories (which is yet too common) – rather than a top-down design cycle, the goals of the protocols were "clear" to the developers and the code was written as to satisfy these clear, yet undocumented, goals. Consequently, much of the effort had been targeted towards obtaining a formal specification of the protocol from its implementation. This effort not only allowed for the formal verification of the protocol, but also as a gave the developer a (much needed) starting point for expansion of the protocol (to the write-once case);

- Many (if not all) formal verification techniques are criticized as not being scalable. We show that, a close collaboration between HPC and verification communities, with one can achieve a "good enough" abstraction where the verification techniques, while not necessarily scalable, are sill useful and applicable for the formal analysis of a "real-life" complex protocol. In fact, many steps of the abstraction were accomplished using such automatic "unscalable" formal techniques;

- Even at the high level of abstraction accomplished, it was possible to detect a bug in the protocol, which may have been the source for erroneous behavior that had been observed. This was reported to the developers and has subsequently been fixed in the implementation.

Our results are promising and we hope our work will encourage closer collaboration between HPC and Formal Verification communities to apply formal techniques to complex, HPC systems as to yield reliable systems. This is of paramount importance as we scale up towards petascale systems.

The paper is organized as follows. In Section 2, we present a brief overview of LambdaRAM and present a step-by-step abstraction of the model. Section 3 describes the underlying theory of the techniques and methodologies used in the formal verification, and Section 4 describes the verification process. In Section 5, we present our conclusions and discuss our future directions. We present overview of LambdaRAM and the formal methods used in the appendices.

2 The Abstraction Phase

In this section we present our abstraction of LambdaRAM. A functional description of LambdaRAM is in Appendix A. LambdaRAM currently supports a Read-Only consistency mode where the original data is never modified. This mode is sufficient for most data-intensive HPC applications [19]. A typical scenario of an application using LambdaRAM is shown in Fig. 1 where a 256-node NASA Ames cluster in California and a 100-node NASA Goddard GSFC cluster in Maryland are interconnected by a dynamically provisionable ultra-fast high-speed optical networks over the National LambdaRail (NLR). The figure depicts a parallel weather prediction application running on the NASA Ames cluster routinely accessing the MERRA data store, which is a multi-terabyte weather data repository located at NASA GSFC. The application uses LambdaRAM for real-time data access to the remote dataset. LambdaRAM encompasses the combined memory of the two clusters and caches (at most 1.4TB of) the MERRA data. It also manages the data access to the entire multi-terabyte MERRA data repository for the application.

To abstract LambdaRAM, we made the following assumptions:

1. HPC clusters are typically heterogeneous. Modeling heterogeneous cluster configuration drastically increases the number of parameters in the model. We have currently assumed homogeneous cluster configuration wherein all the servers and clients have the same system configuration.

2. High-speed optical networks are point-to-point networks. Thus, they do not facilitate all-to-all communication needed for cluster-to-cluster communication. Aggregation technologies in Layer 2 (Ethernet grooming), Layer 3 (Routers), etc. are used to achieve all-to-all communication. Aggregation technologies and the multiple networks paths between any source-destniation pair in optical networks result in reordering of messages. We have currently not considered message re-orderings in our model.

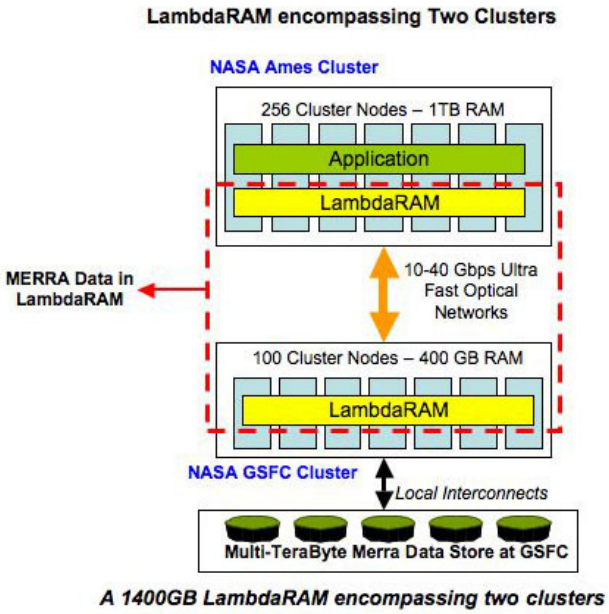

Figure 1. Typical Use-case Scenario Of LambdaRAM

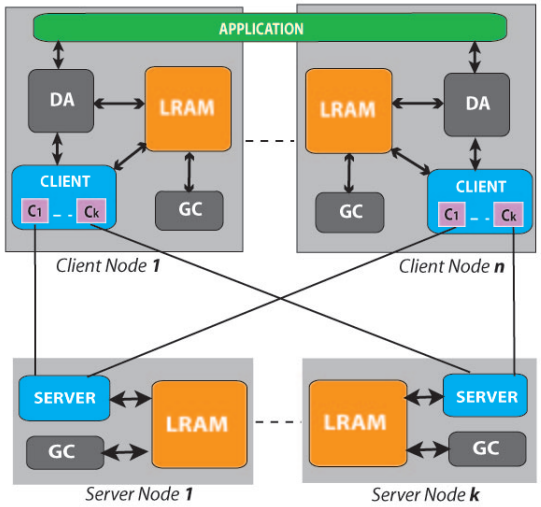

Figure 2. Initial Model Of The LambdaRAM **Protocol**

3. Data-intensive applications are inherently parallel, where each data request is decomposed into requests of data blocks from several nodes. If a node request exceeds the node's memory in a LambdaRAM computation, the request is further decomposed into a sequence of data blocks, each fitting the node's memory. Here, we assume that requests do not exceed the maximum memory available on a single node and bypass the need to model a sequence of requests.
4. LambdaRAM can encompass the memory of multiple clusters interconnected by high-speed networks. We restrict our attention to a two cluster, client-cluster server-cluster configuration, which is one of the common configurations of LambdaRAM.

With these assumptions, we worked closely with the development team to formulate a higher level abstraction of LambdaRAM.

Initial Phase

The Initial abstraction of an application running on two machines is as shown in Fig. 2

The client cluster of LambdaRAM, on which the application typically executes, is composed of the following modules:

Data Access Module (DA): responsible for satisfying an application's request for data blocks. DA first checks if the data block is locally cached, and sends a request to an appropriate client to fetch the block if is not cached.

Client Module (CLIENT): satisfies the DA's request for uncached blocks from remote servers. It consists of a client connection to each server.

Garbage Collector (GC): aids the memory management of the Local LambdaRAM by employing various heuristics including (e.g. LRU, and MRU).

Local LambdaRAM *Cache (LRAM):* a shared data structure on each node, which is part of the global LambdaRAM Cache.

The server-cluster of LambdaRAM is composed of the following modules: Local LambdaRAM Cache (LRAM), Garbage Collector (GC) and the Server Module (SERVER). The LRAM and GC are similar to the client-cluster case, and, the SERVER is responsible for satisfying the data requests from the clients.

Second Phase

We simplified the initial abstraction by assuming that the datasets fit into the combined memory of the LambdaRAM server nodes. This enabled us to eliminate the garbage collector on the server nodes and simplify the server nodes to a single server process serves clients' requests. The garbage collector on the client's side could not be similarly abstracted since assuming the datasets fit into the memory at a client is not realistic. The resulting system is shown in Fig. 3.

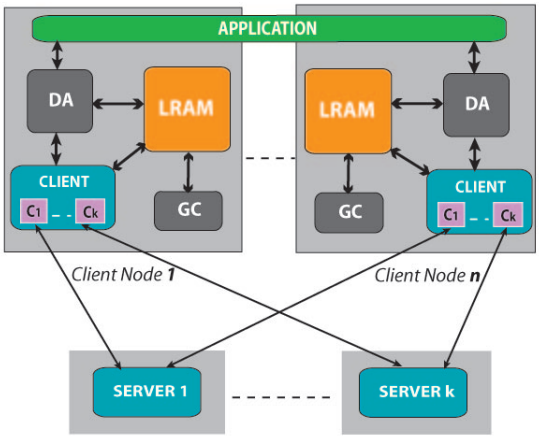

Figure 3. Abstraction With Elimination Of Server Memory Management

Third Phase

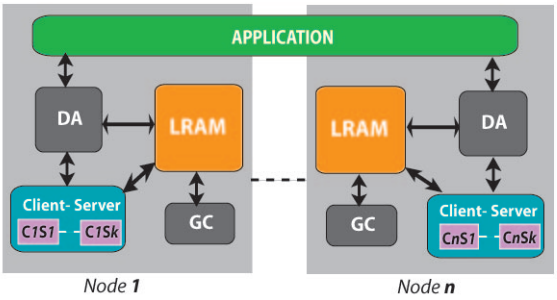

Figure 4. Client-Server Pair Abstraction

LambdaRAM uses reliable transport protocols (TCP, Parallel TCP, Celeritas, etc.) [18] and we assume reliable communication between clients and servers. We combine the client and server modules into a single client-server pair module, shown in Fig. 4 as they exhibit a symmetric behavior for the Read-Only case.

Final Abstraction

Applications that use LambdaRAM are typically data-parallel applications and exhibit a symmetric behavior on each node. The simplified abstraction, utilizing this symmetric behavior, is as shown in the Fig. 5.

Using the abstraction of , Figures 6 and 7 describe the message sequence charts of the main events in the system – Fig. 6 describes the events from the time application requests data blocks until it receives them, and Fig. 7 describes the concurrent (and independent) activity of the garbage collector.

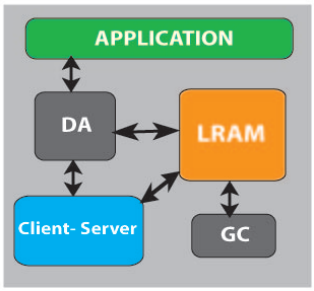

Figure 5. Abstracting The Symmetric Property Of Parallel Systems

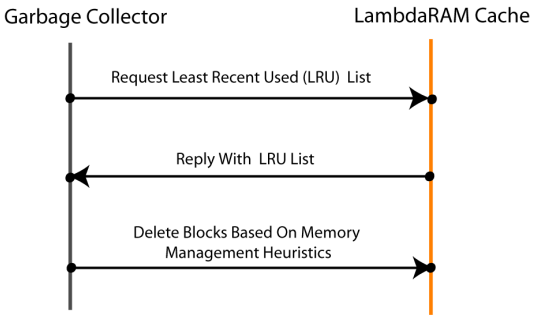

Figure 7. Message Sequence Chart For Garbage Collection

Formal Derivation/Verification of Abstraction

All steps in the abstraction described above were obtained manually. In retrospect, we believe that, exploiting locality, we could have automatically decomposed the system into the GC, Application together with DA, and Clients/Servers modules. This belief is supported by some preliminary tests we performed. Verifying that the Client-Server module of Fig. 5 abstracts the multiple client-server modules of Fig. 2 is attainable in the theorem prover TLPVS of [13], which combines the temporal logic framework with PVS [12], by using refinement mappings similar to those described in [1]. To our knowledge, there is no model-checking based technique to verify the correctness of this abstraction, hence the need for a theorem prover. We did model-check some (not too) small instantiations using TLV [15] and obtained a sanity check for the correctness of this abstraction. We note that the manual proof is rather straightforward.

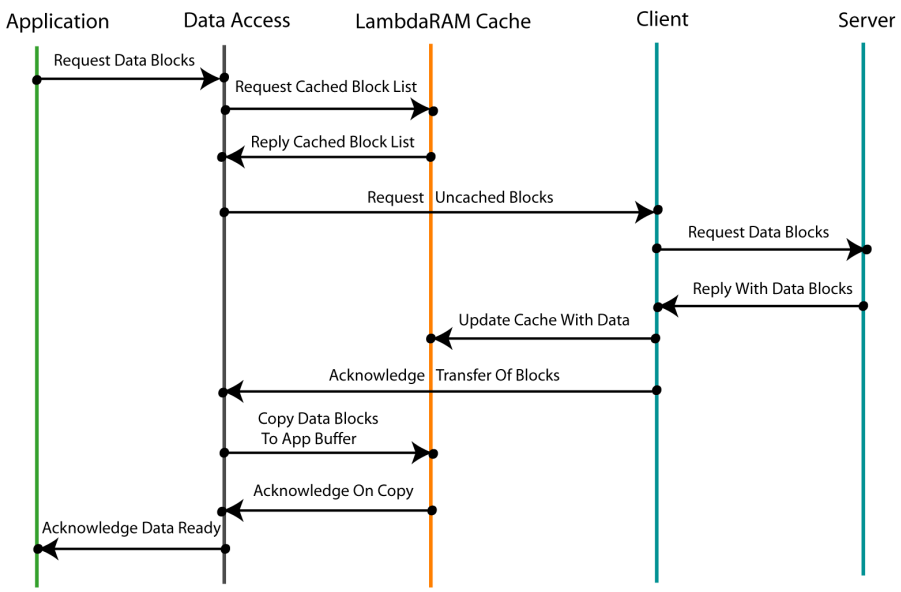

Figure 6. Message Sequence Chart For Treating Application's Requests

3 Formal Techniques and Tools

We present a brief overview of the formal techniques and tools we used.

There are two properties we were requested to verify: a *safety* property (of the type $\Box\, p$ where $\Box$ is the temporal operator "always" and p is a state assertion, i.e., an assertion whose truth depends only on the state it is interpreted on), and a *progress* property of the type $p \Rightarrow\!\!\!\!\succ \Diamond\, q$, where $\Diamond$ is the temporal operator "eventually" and both p and q are state assertion. This property read as "every p-state is eventually followed by a q-state." Since the system is *parameterized* (by, for example, the size of the memory, the bound on the size of cached memory, etc.), each assignment of values to the parameters defines an *instantiation* of the system. Verification of such a system implies verification of *every* instantiation, which, in general, is undecidable. There are, however, several techniques one can use that are sound, that is, if one succeeds verifying the system with these techniques, every instantiation of the system satisfies the properties.

Initial Steps

To make the verification task more manageable we made some simplifying assumptions. See Section 4 for details. With the simplifications, we obtained a single parameter system, the parameter being the size of the memory. We coded the resulting system in SMV, which allows to consider it a *bounded just transition system* (BJTS, see Appendix B for details) – a transition system with justice (weak fairness) assumptions. Having the system expressed as a BJTS allows for analyzing it with several formal techniques as well as to apply some existing symbolic model checking tools on it (see the end of this section).

Verification Techniques

To prove safety ($\Box\, p$ above) we employed the *Invisible Invariant* methodology of [14, 3, 20] which allows for automatic verification such properties for a parameterized system. See Appendix C for an overview of the method.

To prove progress, we use a simplification of the method of [5]: Suppose we want to show that a system that is composed of some parallel modules satisfies a progress property $p \Rightarrow\!\!\!\!\succ \Box\, q$, i.e., every p-state is eventually followed by a q-state. Let $P_1, \ldots, P_n$ be a sequence of all the system's modules, $k > 0$ be a constant, and assume that $1 \leq \ell \leq n$. Let r be the property: "Once p becomes true, if P_1 takes k (not necessarily consecutive steps), then P_2 takes k steps, then $\ldots$, then P_ℓ takes k steps, then a q-state must be reached in this duration." Then, obviously, r implies the progress property $p \Rightarrow\!\!\!\!\succ \Diamond\, q$.

More formally, let M_0 be a new module described in Fig. 8, where active and $\mathit{counter}$ are fresh variables, $\mathit{active} \in \{0, 1..\ell\}$ and $\mathit{counter} : [1..\ell] \mapsto [0..k]$.

Similarly, for each $j = 1, \ldots, \ell$, let M_i be the module described in Fig. 9.

We then have the following theorem, whose proof follows from a similar one in [5]:

If $(active = 0 \ \wedge \ p \ \wedge \ \neg q)$
  then $active := 1$**; for all** $j \in [1..\ell], \ counter[j] := 0$
Elsif (q)
  then $active := 0$
Elsif $((1 \leq active < \ell) \ \wedge \ (counter[active] = k))$
  then $active := active + 1$

Figure 8. The process M_0

If $(active = j \ \wedge \ counter[j] < k)$
  then $counter[j] := counter[j] + 1$

Figure 9. The process M_j

Theorem 1. *If system*

$$(M_0 \ \|\|\| \ \|_{j=1}^{\ell}(P_j \|\|\| M_j)) \ \| \ P_{\ell+1} \| \ldots \| P_n$$

satisfies

$$\Box(active \ \rightarrow \ \bigvee_{j=1}^{\ell} counter[j] < k)$$

then the system $\|_{i=1}^{n} P_i$ *satisfies* $p \Rightarrow \diamondsuit \, q$.

Note that we assumed that each module can always take an idle step. We, however, don't wish to count idle steps when non-idle ones are enabled (which will allow the counters to grow indefinitely, violating justice). In practical terms, we have a single "Idle" module that performs all the idle steps, and our processes, once scheduled, idle only if they have no other option. Note also that the method described applies to proofs where ℓ does not depend on the system parameter N. Hence, this is a simpler situation than the one described in [5].

Theorem 1 demonstrates how progress properties can be transformed into safety properties. While ℓ is independent of N, the progress property may be parameterized, hence, we may need to verify the safety (implied by progress) of the new system using parameterized verification techniques.

Tools

We use TLV [15] for model checking. TLV is a BDD-based model checker that uses SMV for its input language, and has some scripting that make it especially suitable for our purposes.

4 Into the Verification Den

LambdaRAM is implemented in C++; the code base is currently 30,000 lines, which renders it impossible to formally verify by existing automated tool. The abstraction described in Section 2 identified five modules. There were two properties that needed to be formally verified:

1. (Safety) – the number of non-empty cache blocks never exceeds the maximal memory that can be cached (which is given as a parameter);
2. (Liveness) – every requested block is eventually is granted;

These two properties seem like the typical toy properties given in basic formal verification texts, however, in the case of the LambdaRAM code, there are several factors that renders their verification considerably harder: The memory is multi-dimensional, a "block" consists of a list of hyper-boxes "chunks" of the memory, the number of applications, the shape of the memory, the maximal amount of memory that can be cached at a given time, the number of requests an application can issue, as well as numerous other parameters, can all vary, and formal verification should prove (1) and (2) regardless of the value of the parameters[1].

Simplifying Assumptions

We opted to make some simplifying assumptions in order to obtain an initial formal verification, and then to remove the assumptions. The assumption were chosen so as to be independent of one another with respect to verification of (1) and (2). The main assumptions are:

The memory is a linear array. While the complex structure of both memory and requests are an inherent part of the protocol, for proving (1) and (2) it suffices to assume that the requests can be translated to sequences of memory addresses, and that the latter can be represented as absolute addresses over $\mathbb{N}$. At some later point, it may be necessary to verify this translation between the hyper-boxes into a sequence of addresses, but this is irrelevant to the properties we are aiming to verify.

Most parameters can be assumed to be small constants. The parameters that are relevant to proving (1) and (2) are the bound on the maximal number of memory cells that can be cached at a given time (`MaxMemory`), the number of application threads, the number of memory blocks (N), and the bound on the size of requests. Obviously, `MaxMemory` should be larger than the maximal request size. However, as our abstraction of the memory implies, it suffices to assume that the request size is small. To simplify matters, we chose the request size to be 1. For sanity checks, we also verified the protocol with larger request sizes, and, as expected, obtained no new behaviors. Similarly, we chose `MaxMemory`

[1] All TLV code used for the experiments, is in *http://www.evl.uic.edu/venkat/papers/memocode08.html*

to be some multiple of the request size. Again, we experimented with several values, and settled on 2 for the presentation here. As we note in the future work section, we are currently working on obtaining the automatic verification with general parameters, or on formally proving that small values we chose indeed suffice.

Module Abstraction. We chose to (manually) abstract some modules, to verify the system with the abstracted modules, and to separately verify that the abstraction is correct. The latter was accomplished by methods similar to [1]. Since we are using a model checker, we could not prove the abstraction for arbitrary instantiations of parameters, however, we did obtain successful model checking runs with non-trivial instantiations, and a deductive proof that we are now "guiding" the tool TLPVS to generate.

Atomicity assumptions. As is common in this type of parameterized verification, we assumed that some tests are performed atomically while in any reasonable implementation this is not a realistic assumption. We are currently working on applying some of the new methodologies (e.g., [2]) to remove such atomicity assumptions.

Proving Safety

The safety property we wish to prove is that the number of cache blocks that are cached or are in transit never exceeds the maximal memory that can be cached. For each memory block i, the variable $\texttt{CacheBlock\_State}[i]$ denotes the state of the i^{th} memory block, and it is neither cached nor in transit when it equals EMPTY. Hence, the safety property we want to verify is that for every instantiation N,

$$\Box(\sum_{i=1}^{N}(\texttt{CacheBlock\_State}[i] \neq \texttt{EMPTY}) \leq \texttt{MaxMemory})$$

To prove the property, we employed the method of Invisible Invariants using total number of blocks as a the single parameter. As discussed in Section 3, we chose RequestSize to be 1 and MaxMemory to be 2. The transition relation is of the form $\exists i.\forall j.\rho(i,j)$ where i and j range over 1..TotBlocks and $\rho(i,j)$ refers to two free **index** variables. Suppose we are seeking an invariant of the form $\forall i,j.\phi(i,j)$. Using invisible invariants (see Appendix C), we can use instantiation of size $N_0 = 4$. In fact, we chose a larger N_0 and succeeded in generating inductive invariants for shapes that have a 2- and a 3- universally quantified. We approached the problem in two directions – in one, we went the usual invisible invariant way, starting with the set of reachable states, projecting on two (or three) processes, and generalizing onto the others. We also attempted to produce the invariant by starting with Θ, the initial assertion, and iteratively projecting and generalizing it, until a fix point is reached. Surprisingly, both methods produced the same inductive invariants, only the latter (starting with Θ and reaching a fix point) took considerably more time. This is contrary to prior simpler experiences where both methods produced the same invariants and the latter method converged much faster. It's hard to draw conclusion from this, and as much as we can, we'll continue to use both methods simultaneously (if for nothing else, it proved to be a very efficient debugging tool) and attempt to gauge their relative merits. The results are shown in Fig. 10

We ran the experiments on 2.2GHz Intel Core 2 Duo MacBook Pro with 2GB of 667Mhz DDR2 SDRAM. The Darwin Kernel Version running on the MacBook Pro was 8.10.1. TLV 4.18.4 was used for model checking.

Proving Liveness

The liveness property we wish to establish is that every requested block is eventually granted. A block i is requested when $\texttt{RequestBlockList}[i]$ is set, and is granted when $\texttt{RequestBlockList}[i]$ is reset. Hence, the liveness property is that for every $i \in [1..\texttt{TotBlocks}]$,

$$\texttt{RequestBlockList}[i] \Rightarrow \Diamond \neg \texttt{RequestBlockList}[i]$$

Since all the blocks are treated symmetrically, it suffices to establish the property for a *representative* block, say $i = 2$. Hence, we focus on verifying:

$$\texttt{RequestBlockList}[2] \Rightarrow \Diamond \neg \texttt{RequestBlockList}[2]$$

Following the ideas in Subsection 3, we arranged the modules where P_1 is Application (App), P_2 is DataAccess (DA), P_3 is ClientServer (CS), and P_4 is GarbageCollector (GC). Thus $n = 4$. We also choose ℓ to be 3 and k (the counter bound) to be 3. With $p = \texttt{RequestBlockList}[2]$ and $q = \neg\texttt{RequestBlockList}[2]$, we verified the system

$$(M_0 \ \|\|\| \ \|_{j=1}^{\ell}(P_j|||M_j)) \ \| \ P_{\ell+1} \| \ldots \| P_n$$

against the safety property

$$\forall N. \ \Box(active \rightarrow \bigvee_{j=1}^{\ell} counter[j] < k)$$

using the method of invisible invariants (taking the same N_0 as before). From Theorem 1, it now follows that the original system satisfies

$$\texttt{RequestBlockList}[2] \Rightarrow \Diamond \neg \texttt{RequestBlockList}[2]$$

In fact, before the successful verification, we obtained error traces, which allowed each module to take infinitely

N_0	invariant shape	from reachable	from Θ
4	$\forall i, j.\phi(i, j)$	0.45 sec	2.9 sec
5	$\forall i, j.\phi(i, j)$	1.12 sec	7.08 sec
6	$\forall i, j.\phi(i, j)$	2.06 sec	14.66 sec

Figure 10. Run Time Results

many idle steps. A more careful inspection revealed a trivial bug – the garbage collector (GC) was always allowed to change the observable behavior of the system even when there were no changes in the memory since its last pass. This enabled a scheduler that scheduled other modules only when the garbage collector prevented them from taking a "productive" step. The designers have then system to make sure the garbage collector doesn't perform unnecessary work, and we could the prove the liveness property.

5 Conclusion and Future Work

We presented a simplified model of LambdaRAM– a high-performance, multi-dimensional, wide-area distributed cache for data intensive HPC applications – and its protocol for the read-only case. We formally verified the protocol using automated and deductive techniques. Through formal verification, we were able to identify a bug in the cache management of the protocol which has since been subsequently fixed in the implementation.

Our work entailed a significant amount of reverse engineering – faced with a 30K line code, we had to abstract it and determine its correctness criteria. We were fortunate enough to have an extremely cooperative LambdaRAM development team who wished to identify and fix the bugs in the protocol to enable a reliable deployment in NASA's mission-criticial applications. Another motivation was that future expansions (e.g., for the write-once case) will be designed in a top-down manner, that is, the abstract verified model first, then a step-wise process leading in to the complex code.

We are currently working towards incorporating additional parameters including, the k-dimensional memory space with requests for hyperboxes of various dimensions, the multiple applications, multiple datasets, memory quality of service requirements. Since it may not suffice to use automatic techniques to handle such parameters (our small model will possibly be small but not small enough to avoid state-explosion), our plan is to use compositional techniques. For example, our client-server model is an abstraction of a composition of two modules, and, this simplified obtaining an inductive invariant.

In the long term, we plan to build a tool-suite that will allow for verification of similar systems by user-guided combinations and composition of formal verification techniques.

Acknowledgement: We would like to thank Ittai Balaban, Robert Burns, Ariel Cohen, Naveen Krishnaprasad, Ken McMillan, Anand Patwardhan, Amir Pnueli, Luc Renambot, and Mike Seablom for the insightful discussions regarding the specification and verification of the LambdaRAM protocol. This research was supported in part by ONR grant N00014-99-1-0131 and NSF Awards CCF-0742686, CNS-0720525, CNS-0420477, OCI-0225642, OCI-0441094. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proceedings of the Ninth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. Springer Verlag, 2008.

[3] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In CAV'01, pages 221–234. LNCS 2102, 2001.

[4] I. Balaban, Y. Fang, A. Pnueli, and L. Zuck. An invisible invariant verifier. In *Proc.* 17^{th} *Intl. Conference on Computer Aided Verification (CAV'05)*, pages 408–412, 2005.

[5] Y. Fang, K. L. McMillan, A. Pnueli, and L. Zuck. Liveness by invisible invariants. In *FORTE*, pages 356–371, 2006.

[6] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Proc. of the* 5^{th} *conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lect. Notes in Comp. Sci.*, pages 223–238, Venice, Italy, January 2004. Springer-Verlag.

[7] http://gmao.gsfc.nasa.gov/systems/geos5/. The goddard earth observing system model, version 5 (geos-5), nasa, goddard space flight center, 2007.

[8] http://map06.gsfc.nasa.gov. Nasa, goddard space flight center, 2006.

[9] N. Krishnaprasad, V. Vishwanath, S. Venkataraman, A. Rao, L. Renambot, and A. J. J. Leigh. Juxtaview a tool for interactive visualization of large imagery on scalable tiled displays. In *Proc. of IEEE Cluster 2004, San Diego, CA, 09/20/2004 - 09/23/2004(CLUSTER 2004)*, 2004.

[10] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
[11] H. B. Newman, M. H. Ellisman, and J. A. Orcutt. Data-intensive e-science frontier research. *Commun. ACM*, 46(11):68–77, 2003.
[12] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
[13] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.
[14] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.
[15] A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, page 184, New Brunswick, NJ, USA, / 1996. Springer Verlag.
[16] L. Smarr, A. Chien, T. DeFanti, J. Leigh, and P. Papadopoulos. The optiputer. In *Communications of the ACM, Special Issue: Blueprint for the Future of High-Performance Networking, 46(11):58-67, Nov. 2003.*, 2003.
[17] V. Vishwanath, R. Burns, J. Leigh, and M. Seablom. Accelerating tropical cyclone analysis using lambdaram. *Elesvier FGCS, The International Journal of Grid Computing*, 2008, To appear.
[18] V. Vishwanath, T. Shimizu, M. Takizawa, K. Obana, and J. Leigh. Towards terabit/s systems: Performance evaluation of multi-rail systems. In *Proc. 26^{th} IEEE Annual Conference on Computer Communications (INFOCOM'07)*, 2007.
[19] C. Zhang, J. Leigh, T. A. DeFanti, M. Mazzucco, and R. L. Grossman. Terascope: distributed visual data mining of terascale data sets over photonic networks. *Future Generation Comp. Syst.*, 19(6):935–943, 2003.
[20] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages, Systems, and Structures*, 30(3–4):139–169, 2004.

Appendices

A Functional Architecture of LambdaRAM

The functional architecture of LambdaRAM [17] is shown in Fig. 11. LambdaRAM consists of a data access layer that satisfies the data requests of an application. The data access layer interacts with the distributed data cache to satisfy these data requests. The distributed data cache spans the memory of multiple clusters and cluster nodes. If the requested data is not present in the distributed data cache, the data cache layer fetches the data from the storage system using the I/O Abstraction layer. We briefly describe each subsystem.

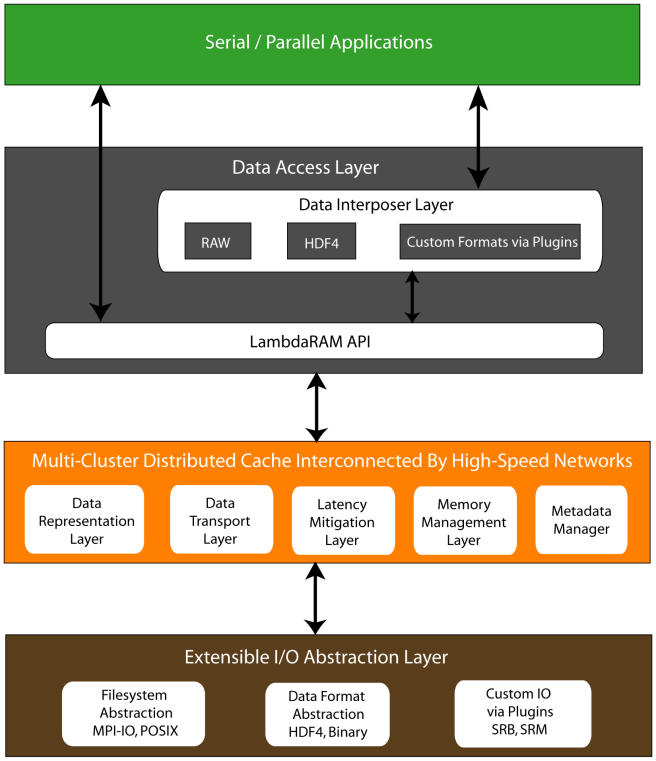

Figure 11. Functional Architecture Of LambdaRAM

Data Access Layer. An application can use either the LambdaRAM API or the data format interposer layer to access data. The data format interposer layer enables an application to use LambdaRAM without modifying a single line of the applications code. The LambdaRAM API provides an intuitive API to access multi dimensional datasets.

Multi-DimensionalDistributed Cache Layer. The Multidimensional Distributed Cache Layer enables efficient access to the data. The data representation layer and memory management layer are responsible for the multi-dimensional cache management over the multiple clusters. The latency mitigation layer helps overcome the network latencies by fetching data just before an application needs it. The data is transferred efficiently between the various levels of caches using the Data Transport Layer.

Extensible I/O Abstraction Layer. The extensible I/O layer enables efficient access to multi-dimensional datasets on storage systems. The Filesystem abstraction layer enables the use of high-performance parallel IO interfaces, including MPI-IO, to efficiently access data on parallel filesystems. The Data format abstraction layer enables ac-

cess to datasets present in scientific data formats.

B Bounded Just Transition Systems

As a computational model for parameterized bounded-data systems we use *bounded just transition systems*, that are a compassion-less variant of the model of *bounded fair transition system* of [6].

B.1 Just Transition Systems

We present a variant of the *just transition system* of [10]. A JTS is described by $S = \langle V, \Theta, \mathcal{T}, \mathcal{J} \rangle$, with:

- V — A finite set of typed *system variables*. A *state* s of the system provides a type-consistent interpretation of the system variables V, assigning to each variable $v \in V$ a value $s[v]$ in its domain. Let Σ denote the set of all states over V. An *assertion* over V is a first order formula over V. A state s satisfies an assertion φ, denoted $s \models \varphi$, if φ evaluates to TRUE by assigning $s[v]$ to every variable v appearing in φ. We say that s is a φ-state if $s \models \varphi$.
- Θ — The *initial condition*: An assertion characterizing the initial states. A state is called *initial* if it is a Θ-state.
- $\mathcal{T}$ — A finite set of transitions. Every transition $\tau \in \mathcal{T}$ is an assertion $\tau(V, V')$ relating the values V of the variables in state $s \in \Sigma$ to the values V' in an $\mathcal{D}$-successor state $s' \in \Sigma$. Given a state $s \in \Sigma$, we say that $s' \in \Sigma$ is a τ-*successor* of s if $\langle s, s' \rangle \models \tau(V, V')$ where, for each $v \in V$, we interpret v as $s[v]$ and v' as $s'[v]$. We assume an *idle* transition $\tau_I = \bigwedge_{v \in V} v = v'$.
- $\mathcal{J}$ — A set of assertions over V. A computation should have infinitely many J states for every $J \in \mathcal{J}$.

Let $\sigma : s_0, s_1, s_2, \ldots$, be an infinite sequence of states. We say that σ is a *computation* of S if it satisfies the following requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, ...$, state $s_{\ell+1}$ is a τ-successor of s_ℓ for some $\tau \in \mathcal{T}$.
- *Justice* — for every $J \in \mathcal{J}$, there are infinitely many positions $k \geq 0$, such that s_k is a J-state.

Composition of Just transition Systems Assume two JTS's $S_1 \colon \langle V_1, \Theta_1, \mathcal{T}_1, \mathcal{J}_1 \rangle$ and $S_2 \colon \langle V_2, \Theta_2, \mathcal{T}_2, \mathcal{J}_2 \rangle$. The *asynchronous parallel composition* of S_1 and S_2, denoted by $S_1 \| S_2$, is the JTS

$$\big(V_1 \cup V_2, \Theta_1 \wedge \Theta_2, \mathcal{T}, \mathcal{J}_1 \cup \mathcal{J}_2\big)$$

where $\mathcal{T}$ includes, for every $i = 1, 2$, all the transitions in $\tau \in \mathcal{T}_i$ with the conjunct $\bigwedge_{v \in V_i - V_{3-i}} v = v'$. That is, each transition of $S_1 \| S_2$ is a transition of one of its components requiring the transition to alter no value not in its component's domain.

The *synchronous parallel composition* of S_1 and S_2, denoted by $S_1 \| \| S_2$, is the JTS

$$\big(V_1 \cup V_2, \Theta_1 \wedge \Theta_2, \bigvee_{\tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2} \tau_1 \wedge \tau_2\big)$$

B.2 Bounded Just Transition Systems

To allow the application of the invisible invariants method, we further restrict the systems we study, leading to the model of *bounded just transition systems* (BJTS). For brevity, we describe here a simplified two-type model; the extension for the general multi-type case is straightforward.

Let $N \in \mathbf{N}^+$ be the *system's parameter*. We allow the following data types:

1. **bool**: the set of boolean and finite-range scalars;
2. **index**: a scalar data type that includes integers in the range $[1..N]$;
3. **data**: a scalar data type that includes integers in the range $[0..N]$; and
4. Any number of arrays of the type **index** $\mapsto$ **bool**. We refer to these arrays as *boolean arrays*.
5. At most one array of the type b : **index** $\mapsto$ **data**. We refer to this array as the *data array*.

Atomic formulas may compare two variables of the same type. E.g., if y and y' are **index** variables, and z is an **index** $\mapsto$ **data** array, then $y = y'$ and $z[y] < z[y']$ are both atomic formulas. For z : **index** $\mapsto$ **data** and y : **index**, we also allow the special atomic formula $z[y] > 0$. We refer to quantifier-free formulas obtained by boolean combinations of such atomic formulas as *restricted assertions*. As the initial condition Θ, we allow assertions of the form $\forall \vec{i}.u(\vec{i})$, where $u(\vec{i})$ is a restricted assertion. As the transitions $\tau \in \mathcal{T}$, we allow assertions of the form $\tau(i)$: $\forall j.\psi(i, j)$ for a restricted assertion $\psi(i, j)$.

B.3 Example of a BJTS

Consider program MutSem in Fig. 12, which is a simple mutual exclusion algorithm that guarantees deadlock-freedom access to critical section for any N processes.
In this version of the algorithm, location 0 constitutes the non-critical section which a process may non-deterministically exit to the trying section at location 1. Location 1 is the waiting location where a process waits until

$$
\begin{array}{ll}
V: & \left\{ \begin{array}{ll} \pi: & \textbf{array}[1..N] \textbf{ of } [0..3] \\ t: & \textbf{bool}; \end{array} \right. \\
\Theta: & \forall i : \pi[i] = 0 \ \wedge \ t = 1 \\
\mathcal{T}: & \left\{ \begin{array}{lll} \tau_0(i): & \forall j \neq i: & \pi[i] = 0 \wedge \pi'[i] \in \{0,1\} \wedge \mathit{pres}(\{\pi[j], t\}) \\ \tau_1(i): & \forall j \neq i: & \pi[i] = 1 \wedge t = 1 \wedge \pi'[i] = 2 \wedge t' = 0 \wedge \mathit{pres}(\{\pi[j]\}) \\ \tau_2(i): & \forall j \neq i: & \pi[i] = 2 \wedge \pi'[i] = 3 \wedge \mathit{pres}(\{\pi[j], t\}) \\ \tau_3(i): & \forall j \neq i: & \pi[i] = 3 \wedge \pi'[i] = 0 \wedge t' = 1 \wedge \mathit{pres}(\{\pi[j]\}) \end{array} \right.
\end{array}
$$

Figure 13. BJTS **for Program MutSem**

$$
\begin{array}{l}
\textbf{in} \quad N \ : \ \textbf{natural where } N > 1 \\
\textbf{local} \quad t \ : \ \textbf{bool where } t = 1 \\
\left\| _{i=1}^{N} P[i] :: \right. \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} 0: & \textbf{NonCritical} \\ 1: & \textbf{when } t = 1 \textbf{ do } t := 0 \\ 2: & \textbf{Critical} \\ 3: & t := 1 \end{array} \right] \end{array} \right]
\end{array}
$$

Figure 12. Program MutSem

the token (t) is available and then takes it. Location 2 is the critical section, and location 3 is the exit section where the process returns the token. As we show, the program guarantees that if some processes are waiting to enter the critical section, eventually some process will succeed. Fig. 13 describes the BJTS corresponding to program MutSem.

C The Method of Invisible Invariants

See [4] for a description of a tool that implements the method as well as a citation list.

The method of invisible invariants generates candidate inductive invariant for a given parameterized system, and checks whether the candidate invariant is inductive and whether it implies the safety property one wishes to prove holds over the system. The generation of the candidate invariant is accomplished by instantiating the system to some small number, and projecting the set of reachable states on a small number of processes, and generalizing it to an arbitrary number of process. For example, when seeking an invariant of the type $\forall i, j.\phi(i, j)$ ($\phi(i, j)$) is quantifier free, the system is generated onto two processes, say i_1 and i_2, and generalized from then into arbitrary i and j. The process of generating the candidate invariants is called *project&generalize*.

After the candidate invariant is generated, it needs to be checked for inductiveness (that is, being implied by the initial states and being preserved by every step) and for implying the property safety that is the goal of the verification. Similarly to *project&generalize*, the verification of inductiveness can be accomplished by symbolic techniques. Since the generated candidates are A-formulae (i.e., when in prenex normal form, the only quantification is universal), the most difficult premise to prove is the inductiveness, which is an AE-formula. As the following theorem (first stated in [14] and later extended in, e.g., [3]) establishes, verification of AE-formuale has a small model theorem property:

Theorem 2 (Small model property)**.** *For every AE-formulae over a* BJTS*, there exists an N_0 such that φ is valid iff it is valid over all instances $S(N)$ for $N \leq N_0$.*

In practice, the theorem implies that in order to verify the inductiveness of a A-candidate of the form $\forall i_1, \ldots, i_k \phi(i_1, \ldots, i_k)$ over a BJTS, it suffices to verify that the formula holds over instantiation of size $N_0 = k + H$, where H is the number of variables under existential quantification in the transition relation (which is an EA-formula), and conclude that the candidate is inductive over instantiation of size $N \geq N_0$.

When applying the method, we usually choose some N' which is at least as large as the computed N_0, and use the same instantiation for both obtaining the candidate and verifying its inductiveness. See, e.g., [4] for details.