H-0272 (I0911-015) November 21, 2009
Computer Science

IBM Research Report

An Ontology and Constraint Based Approach to Cache Preloading

Rajiv Bhatia
IBM Systems and Technology Group
Austin, TX USA

Eyal Bin, Eitan Marcus, Gil Shurek
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

An Ontology and Constraint Based Approach to Cache Preloading

Rajiv Bhatia
IBM Systems and Technology Group
Austin, Texas
rajiv.bhatia@gmail.com

Eyal Bin
IBM Research labs in Haifa, Israel
bin@il.ibm.com

Eitan Marcus
IBM Research labs in Haifa, Israel
marcus@il.ibm.com

Gil Shurek
IBM Research labs in Haifa, Israel
shurek@il.ibm.com

ABSTRACT

The verification of modern microprocessor-based systems requires stressing the cache hierarchy and effectively covering its huge state space. Cache hierarchy initialization (or preloading) is a technique that enables simulation to start from a rich, complex system-level setup, thereby simplifying the task of dynamically driving the hierarchy into the required corner cases.

In this paper we introduce CacheLoader, a new, design-independent cache-preloading technology. The tool's architecture follows the principles of ontology-based software to achieve complete separation between the cache-preloading engine and design dependent knowledge. Constraint satisfaction techniques are used to generate valid, interesting system initialization, and to satisfy explicit user directives. CacheLoader is currently being used by verification teams of several large scale designs in IBM. Results show that this technique provides superior coverage and user controllability, speeds up the construction of mature verification environments, simplifies maintenance, encourages encapsulation of domain knowledge, and enables reuse across verification environments and cache hierarchy designs.

1. INTRODUCTION

Modern microprocessor systems use caches to improve access time for the core's memory operations. Typically, a multi-processor system employs a snoop-based broadcast protocol to maintain memory sub-system cache coherency and data consistency [12]. A MESI cache coherency protocol with only four different states is the basic coherency protocol. Many derivatives of MESI exist, often requiring considerably more states. The POWER6™ scheme [21] for example has thirteen different states that provide an increased level of information to help coherency participants determine how to use or share their copy of the cache line. This added complexity introduces many new scenarios that need to be verified. Consider, for example, the following POWER6 case: A *node* is a physical partition of the system comprising several processor cores and a memory controller managing a chunk of the system's globally shared memory. When a cache-line is held in a valid state in a cache on node A, while its memory space is managed by node B, the cache protocol requires that either the memory controller on node B holds an indication that a valid copy of this line is held on a remote node, or that a copy of this line is held in a cache on node B and tagged with a unique invalid state (*Ig*). It is important for verification to cover scenarios starting from the extreme system state where no indication is held by the home memory controller, and where only a single *Ig* copy, held by a cache on node B, protects system-level coherency for this line.

A common method to verify a cache hierarchy is to construct a random biased testbench and send random—yet legal—read, write, and synchronization transaction requests to the cache and memory subsystem to exercise both realistic and unrealistic workload patterns. This approach is insufficient, especially in systems with complex coherency protocols. Stressing the system requires very long test programs to gradually build the required conditions in the system's cache hierarchy. It is extremely difficult to construct programs that reach the intended system's state in a predictable way. Moreover, reaching such a state would consume many simulation cycles.

A known method to overcome these difficulties is to preload the system with a coherent and interesting initial state before simulation starts. A special cache-preloading module is usually built to achieve this goal. This module should be able to satisfy complex coherency rules, e.g., an address can be tagged as 'modified' in at most one cache entry in the system. In addition, the pre-loading module should provide biasing options to direct the initialization towards interesting and potentially rare corner cases. The verification engineer should be provided with control parameters to enable the creation of various initialization profiles. A good example of the necessity of this capability is a bias that tunes the distribution of the caches' states.

This bias is associated with parameters that provide distribution weights for each state.

In this paper we describe CacheLoader—a new, design–independent, cache-preloading technology. The tool's architecture follows principles of ontology-based software to achieve complete separation between the cache-preloading engine and design-dependent knowledge. Supported designs can vary in cache structure, cache behavior, and coherent initialization rules, as well as in topology and connectivity. Constraint satisfaction techniques are used to process these structural and relational rules as well as encapsulated knowledge of verification experts and explicit user directives to generate a huge variety of valid and interesting initializations for the cache hierarchy.

Experience shows that this technique provides superior coverage and user controllability, speeds up the construction of mature verification environments, simplifies maintenance, encourages encapsulation of domain knowledge, and enables reuse across verification environments and cache hierarchy designs.

The paper is organized as follows. In Section 2 we discuss the underlying technologies, ontologies, and constraint satisfaction, and their application to the cache-preloading domain. Section 3 describes CacheLoader's architecture. Section 4 presents the selected approach for modeling a cache subsystem. Section 5 describes the CSP solution scheme and Section 6 provides our experimental results and concludes with a discussion.

2. ONTOLOGY-BASED, CSP-POWERED

The CacheLoader tool is designed as an ontology-based application [1][9][16], that couples a cache-preloading engine, which is oblivious to the architecture and design details of the specific System-Under-Test (SUT), with a structured, declarative model of system-dependent information.

A key ingredient driving an ontology-based application is a domain specific—in this case cache-preloading specific—modeling language. The language provides the terminology and the basic constructs required to build an SUT-specific ontology. The SUT-specific knowledge is then interpreted by the cache-preloading engine, which in turn uses it to provide the required SUT-specific services.

The ontology does not provide a comprehensive description of the SUT. It only covers aspects of the SUT and its verification environment that are necessary to generate valid, high quality cache initializations, and to interact correctly with the verification environment. The model provides a structural description of the caches in the system such as the size and dimensions of every type of cache, the structure of cache entries, legal states, and legal values of the various directory fields. In addition, the model specifies rules and relations, like coherent initialization rules and hashing functions that associate addresses with the respective congruence classes. A second component of the ontology holds system-specific expert guidance, generally referred to as Testing Knowledge (TK) [1]. TK indicates effective methods to accomplish the application's task (e.g., bug-prone areas and setups to guide or bias cache initializations). This component also includes any per-SUT tuning of the tool's operating principles.

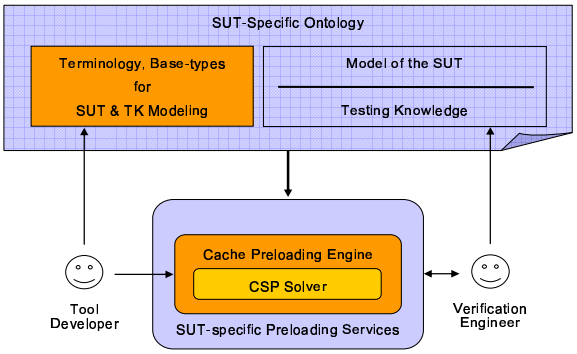

Figure 1. Ontology-based, CSP-powered Cache Loader

The model specifies a very large number of relations between cache entries and fields in the system, reflecting architectural rules and biasing rules that originate from the TK. These relations are naturally modeled as constraints. The use of constraints to specify stimuli generation tasks—representing such tasks as Constraint Satisfaction Problems (CSP)—is not new [6]. It has been shown that CSP solvers could efficiently serve as generation engines by randomly sampling the solution space.

The cache-preloading engine interprets the SUT-specific model according to the terminology and basic constructs provided by the modeling language. A CSP (or a set of CSPs) is then constructed, and handed to a CSP solver to create a valid, biased-random initialization of the system's cache hierarchy.

Figure 1 illustrates an adaptation of the ontology-based scheme to the domain of cache preloading. The verification engineer is not only the end user but also the knowledge expert who develops the ontology and adapts it to design changes. The tool developer supports and enhances the generic engine as well as the basic terminology required for modeling.

In addition to the cache preloading application described in this paper, the ontology-based approach has been successfully adopted by several hardware verification tools including unit, processor, and system-level test generators and post-silicon exercisers [1][2][3].

In summary, the ontology-based approach drives the generalization of domain-wide expert-knowledge and its encapsulation in a reusable engine. It also forces modularity between domain-wide knowledge and application-specific details. It requires the formulization of a standard, domain-specific modeling language that captures the fundamental domain concepts and terminology and encourages the use of declarative modeling techniques for knowledge representation.

Ontology-based tools replicate the impact of a domain expert across different SUTs, assist in accumulating domain-knowledge from one SUT to another, simplify maintenance, and boost standardization and reuse, including reuse of ontology modules.

The alternative to an ontology-based approach, and one that is commonly deployed in the industry, is the testbench development process which uses specialized languages and follows strict methodologies [11]. Development environments supporting these languages and methodologies are available from leading design automation tool vendors [10][19]. Testbench modules are typically built for a specific architecture, protocol or SUT, and have deep knowledge about the SUT embedded in their code. Most testbench components do not provide good (or even any) separation of the application-specific knowledge from the core service engine. Also, testbench platforms do not provide the user with effective mechanisms to deeply parameterize the model and enforce the described layering of domain knowledge. As a result, it is almost impossible, in the context of a traditional testbench, to achieve the level of portability and reuse between different designs enabled by the ontology-based scheme.

2.1 Ontology Modeling Platform, Constraint-Satisfaction Engine

The ontology-based, CSP-powered scheme and the induced tool architecture call for a modeling platform to support the construction of SUT-specific ontologies and for a powerful constraint satisfaction engine. We use ClassMate, an in-house modeling platform designed to support ontology-based tools, and Generation-Core, an in-house CSP solver designed to handle hardware-stimuli and hardware setup problems.

ClassMate is a type-based, ontology modeling platform. As such, it exhibits similarities to some frame-based systems [13], other ontology modeling platforms [8][17], and layers of UML2 [18]. The target domain is modeled as a taxonomy-hierarchy, using a powerful type definition language. ClassMate has a number of features that make it particularly attractive for creating (and maintaining) hardware models for stimuli generation. These include native support for constraints between objects and between subcomponents of objects; a powerful type refinement mechanism that goes well beyond classical inheritance; packages that allow controlled redefinition of types for follow-on designs; and a rich set of extended data types, including collections, meta-types, and bit-vectors for handling such things as arbitrarily sized addresses and data values. ClassMate provides a graphical studio (see Figure 3) for constructing, browsing, editing, and refactoring the model.

The Generation-Core [15] deploys a MAC-based [4][7] algorithm to manipulate and solve a network of constraints, where each constraint is represented by a constraint-propagator routine. Constraint networks may be dynamically constructed and modified to enable, for example, problem partitioning by abstraction. The solver is designed to support hardware-stimuli and hardware setup problems; it supports well distributed sampling of the solution space, very large variable domains, bit-wise operators, soft-constraint hierarchies, conditional problems, and approximated propagators.

Alternative techniques for solving constraint problems were rejected because of the lack of expressive power of their modeling language. Non-boolean variables and non-linear constraints that are needed by CacheLoader (See Figure 4 below) are difficult to express in SAT [14] and ILP **Error! Reference source not found.**.

3. CACHELOADER'S ARCHITECTURE

Figure 2 is a schematic block diagram representing the various components, inputs, and outputs of CacheLoader. At the center, the tool's core is a design-independent engine that reads the ontology, the system's topology, and user initialization requests to produce biased-random initialization for the systems' cache hierarchy. CacheLoader's core encapsulates a CSP solver as its prime generation engine. The tool's core connects to the surrounding components and inputs through a design-independent interface.

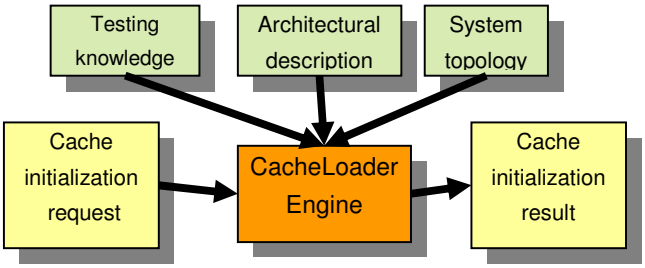

Figure 2. CacheLoader architecture

The ontology comprises an architectural description component and a testing knowledge (TK) component. As mentioned before, the architectural description component

of the ontology holds structural and behavioral information about the SUT, the structure of the various cache types and memory controllers, hash functions, coherent initialization rules, and more.

Based on this description alone, CacheLoader would have produced random initializations that uniformly cover the legal state space. The role of the TK, the second component of the ontology, is to direct or bias the engine towards important corner cases that would otherwise have only a low chance of being hit. TK is therefore implemented as a collection of biasing options. Testing knowledge may be design-independent or design-specific. Design-specific TK may sometimes be shared by a family of similar designs. Examples of design-independent TK include distributing the number of initializations of each preloaded address across the cache hierarchy and filling a whole congruence-class (set) with the same or with different addresses. Examples of design-dependent TK include cases such as the single *Ig*-state described in the introduction. These biasing options have parameters that control their activity and tune their behavior; for example, parameters specifying whether to avoid generating a specific case or intentionally generate it, or distribution weights for a biasing option controlling the distribution of cache states in the system. Section 4 provides additional insights about the ontology component, presenting our approach to ontology modeling.

The verification engineer provides the cache initialization request. This input conveys the user's request to follow a specific initialization profile for the current simulation session. The user feeds CacheLoader a pool of addresses to initialize, and activates or tunes the available biasing options. Tuning is achieved by overriding the default parameter values specified by the TK component. The user may override parameter values for the whole SUT, or just for a specified scope. Scopes are designated using system partition names (e.g., node and processor IDs), by specifying cache types, or symbolically. Using symbols, CacheLoader may be guided to randomly choose the sub-system to which a unique set of parameter values should apply. This feature allows the creation of design-independent and configuration-independent requests.

The system topology describes the configuration of a specific model of the SUT for which initialization is now required. It specifies the number and types of caches in the system and their connectivity.

The output of CacheLoader—the cache initialization result—is a list of cache entries and cache lines that should be loaded by the verification environment to various caches and memory locations in the simulation model, to reflect the new initialization**.**

CacheLoader is an essential component of the unit, core, and system verification flow. A test program generator such as IBM's Genesys-Pro [1] generates memory initializations for both instructions and data. These initializations are transferred to CacheLoader to initialize the cache hierarchy. CacheLoader is implemented as a building block that has been integrated into various verification tools.

4. ONTOLOGY MODELING

The ontology that drives CacheLoader holds a description of the SUT (both the architecture and the relevant design details) and a testing-knowledge component. This is not a comprehensive description; it only holds details that are required to generate valid, high-quality, cache initializations.

A specialized hierarchy of modeling building-blocks has been created for CacheLoader, establishing a modeling terminology designed to serve the cache-initialization problem. A system-specific ontology is then constructed by refining these pre-defined building blocks. In addition, a hierarchy of model packages is supported by ClassMate to enable sharing and reuse between similar or derivative designs.

We list here some of the basic terms used for modeling a system, giving special attention to constraints and biasing options. The various types and constraints are depicted in ClassMate's visual studio as shown in Figure 3. The HierarchyPartition base-type specifies the system's structure, while StorageEntity specifies memories and caches. The internal structure of a cache is described using the terms CongruenceClass, CacheEntry, and Field.

Constraints specify architectural rules and implement biasing options. They may be attached to any type and to entities within a type. Constraints may restrict domain values for primitive entities (e.g., the address space managed by a memory controller), or define relations between entities (e.g., coherency rules over congruence-classes).

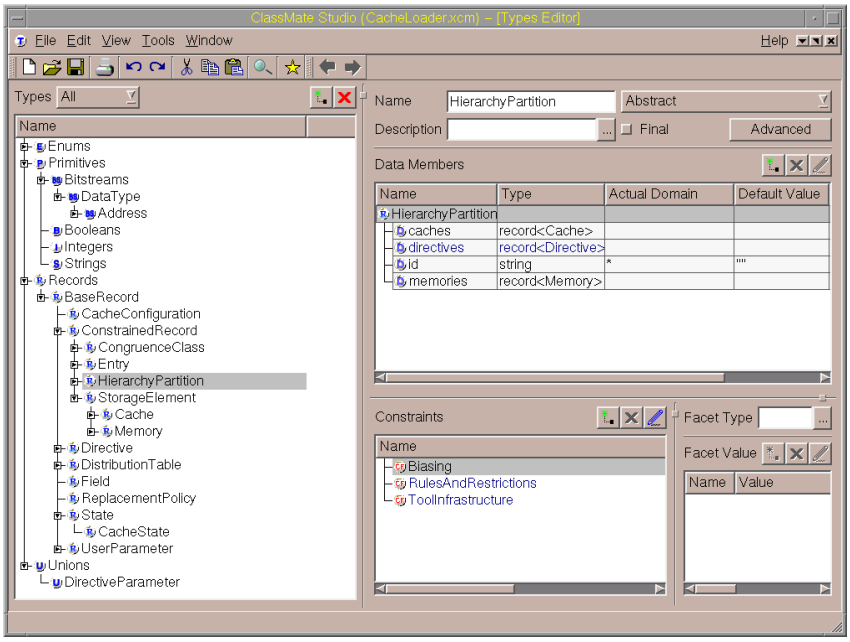

Figure 3. ClassMate's type editor listing CacheLoader base types

There are three types of constraints: (1) RulesAndRestrictions constraints that implement architectural rules (e.g., congruence-hash and coherent-initialization), and simulation-environment restrictions, (2) Biasing constraints that bias cache initializations towards 'interesting', bug prone setups, and (3) ToolInfrastructure constraints that are part of the infrastructure of CacheLoader. In general, the constraints needed to capture the various relations between objects in CacheLoader's model can be very complicated. It is not uncommon for a constraint to include arithmetic, set and field operations, that are logically connected and existentially and universally quantified.

```
1 forEach(addr, addressPool,
2     ($directives.singleLocalIgValidRemote.activation = true) ->
3       (
4         ($directives.singleLocalIgValidRemote.value = true) =
5         (
6           numOf(i, "{L2}", $i.address = $addr and
7                    $addr member_of $i.localAddresses) = 1 and
8           numOf(i, "{L2}", $i.address = $addr and
9                    $addr member_of $i.localAddresses and
10                   $i.logicals.state = L2_Ig ) = 1 and
11          numOf(i, "{L2}", $i.address = $addr and
12                   not $addr member_of $i.localAddresses and
13                   not $i.state member_of << L2_I, L2_Ig, L2_In, L2_Id >> ) > 0
14        )
15      )
16    );
```

Figure 4. A parameterized definition of a biasing option

Directives are the mechanism used to model the biasing options provided to the user. Each directive type declares a new biasing option. A directive includes a list of *DirectiveParameters*, specifying the user-parameters provided for this biasing option. A DirectiveParameter can either take a fixed value or an expression that represents a distribution of values. Each parameter has a default value that the user can override using the cache initialization request. The actual semantics of a directive is specified through *biasing* constraint. The constraint expression provides a declarative, parameterized specification of the biasing goal by relating directive parameters to other entities of the model.

Figure 4 shows a declarative, parameterized definition of a biasing option that supports the 'single *Ig*-state protects coherency' case described in the introduction. The first parameter (line 2) determines whether the biasing option should take effect, while the second (line 4) determines whether the scenario should be created or prevented. Note that the scenario takes effect if and only if exactly one of the L2 cache entries in the local domain holds the address (lines 6-7), it is in the *Ig* state (lines 8-10), and at least one cache which is not in the local domain holds the address in a valid state (lines 11-13).

5. SOLUTION SCHEME

As stated in Section 2, CacheLoader uses CSP formulation to capture the cache initialization problem and then utilizes a CSP solver to generate a valid, biased-random initialization for the system's cache hierarchy. This section provides additional details on this process.

To solve the cache-preloading problem, CacheLoader engine first partitions the problem into several independent sub-problems. This is a performance optimization phase, enabling the solver to handle smaller CSPs, one at a time. The partition is done based on the addresses that need to be initialized and the congruence-class hashing functions of the caches in the system. Each cache specifies a hashing function that defines the mapping between an address and a congruence class in this cache that hosts its respective cache line. Two addresses are dependent if and only if they may be hosted in the same congruence class of at least one cache in the system. An independent CSP is constructed and solved per each transitive closure of dependent addresses.

After partitioning, a CSP is created for each sub-problem. This CSP has variables that represent the physical and logical fields of the caches such as the address of each cache-entry, its state (MESI or its derivatives), the tag, and other design-dependent fields. The CSP is populated with all the constraints (hard and soft) coming from the architectural description, from the testing knowledge, and from the specific cache initialization request provided by the verification engineer.

Each CSP is then handed to the CSP solver and the solutions are held in CacheLoader's repository. When all the sub-problems are resolved, the merged result is forwarded to the verification environment for initialization.

6. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we compare the ontology-based, CSP-powered CacheLoader to a traditional implementation, developed as a module of a C++ testbench. The description of this reference cache-preloading application is followed by a quantitative comparison of the two applications, using coverage tasks and runtime performance as our metrics. The evaluation was done as part of [5]. We conclude with a qualitative discussion analyzing other characteristics of these solutions, demonstrating strengths and weaknesses of each.

6.1 A Reference Cache-Preloading Application

The reference application is a module of a C++ testbench. It implements an exhaustive search algorithm, working on an address-by-address basis. A given pool of addresses is considered in a random order to ensure no ordering biases are introduced. For each address, the solver makes a random number of attempts to initialize that address to a different cache location. For each initialization, the solver picks valid cache and cache state combinations from a prepared list. The candidate initialization is then checked against the current committed list of initializations for compatibility. An initialization is deemed incompatible if it violates a mandatory legality rule, such as a coherent initialization rule, the obvious 'congruence-class-full', or any limitation of the verification environment. These legality rules are hard-coded as predicates, and are mostly design-dependent. If the candidate initialization is compatible, it is added to the committed list. If not, it is rejected and will not be tried again for the current address. The process halts when all addresses in the pool have been visited and evaluated.

The engine is controlled by parameters. For example, probabilities can be assigned to each initialization combination and the number of initializations per address is also controllable. These parameterized policies—or biasing options—are supported by dedicated code. Hence, it requires tool developer expertise and recompilation to enhance the tool with new biasing options.

6.2 Quantitative Results

To evaluate the coverage potential of the two approaches and the controllability of the tools, we defined new coverage tasks that were not previously targeted or measured by the tools. We tested a relatively large 32 cache system; addresses were mapped to 10 different congruence classes with 64 addresses per congruence class. To obtain results for the reference tool, we ran many experiments to find a "best case" result for each task. For the CSP-powered CacheLoader, we demonstrated the capability of adding new biasing options. For both tools, we did not change the tools' code. Later in this section we discuss the option of providing ad hoc code-level support for these tasks in the reference tool.

A verification engineer specified the first three coverage tasks to close specific verification gaps. The first task involves the single *Ig*-state described in the introduction and in Section 4. The second task entails a similar scenario, where a single bit in the memory controller replaces the single *Ig* state. The third coverage task targets preloading configurations, where two parts of the POWER6 coherency protocol disagree. In all these cases, the newly created biasing option was set to target three specific values for the coverage goal: 0%, 50%, and 100% of the addresses. The results are displayed in Figure 5.

For these three tasks, the best results of the reference tool are 9%, 17% and 9% of the addresses, respectively. In addition, we observed that cache initializations generated by the reference tool did not deviate significantly from these results (+/- 5%). The CSP-powered CacheLoader hit 0%, 0% and 8% for the different tasks when it was set to 0%. It generated 33%, 44% and 46% when targeting 50% and finally 84%, 90% and 99% for the 100% goal.

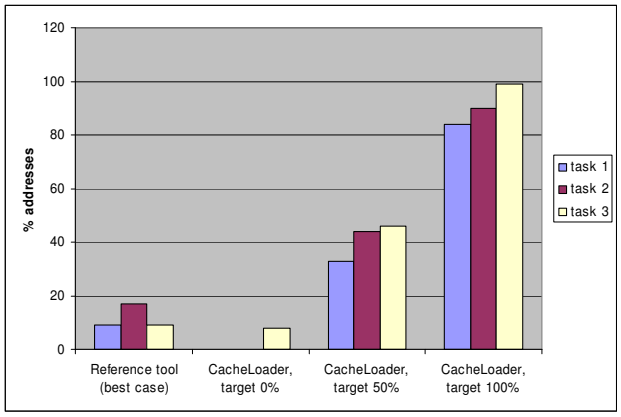

Figure 5. Hitting coverage tasks

The developer of the reference tool investigated how easy it would be to augment the reference tool to include controls that can assist in targeting these coverage tasks. The developer found that it was unclear how to best combine the need for off-node initializations and a specific type of local-node initialization using the current scheme. A satisfying solution does not seem to exist.

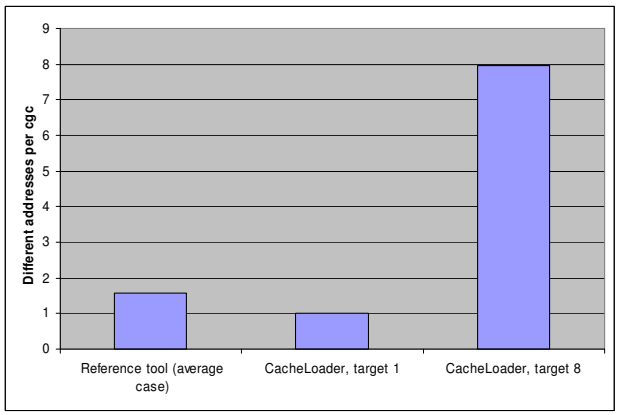

Figure 6. Targeting the number of unique addresses per congruence class

The fourth task aimed to achieve a good distribution of the number of different addresses initialized per congruence class. For the reference tool, we found that the average result was 1.55 addresses per congruence class on the experimental setup. For the CSP-powered CacheLoader, a dedicated biasing option repeatedly achieved exactly 1 address per congruence class when this was the targeted task and 7.97 on the average when the goal was to fill up congruence classes (8 entries for the L2 cache of Power6) with 8 different addresses (Figure 6)

To compare runtime performance, we built two system models and used various address pool sizes. Runtime values were gathered for each test setup on an Intel Xeon 3.06 GHz processor running Linux. Both 32-cache and 64-cache models were run with address pools mapped to 10 and to 50 congruence classes, with variants of 8, 16, 32, and 64 addresses per congruence class. We found that the reference tool runs five times faster on average than the CSP-powered CacheLoader. Since cache-preloading consumes about 1% of the total simulation runtime, this reduction in runtime was found to be acceptable.

6.3 Qualitative Discussion

Like other CSP-based applications, CacheLoader decouples the description of the problem from the solving engine. CacheLoader's domain specific modeling language facilitates this decoupling by enabling a verification engineer—who is not a CSP expert—to model the required knowledge. The derivation of a CSP formulation of the cache initialization problem is fully automated by the tool's core. In contrast, the reference tool interweaves a structural model of the system, the associated coherency rules, design specific biasing options, and a solving procedure into a single software module. Moreover, the algorithm implemented by the reference tool is tied into the details of the specific SUT. This leads to high development and maintenance costs, and low reusability.

A cache-preloading module is a verification IP [11] providing a well-defined service capable of supporting verification environments at various levels of integration (e.g., unit, subsystem, system). As a verification IP, CacheLoader provides a higher level of portability than the original reference tool. Field experience shows that absorbing on-going design changes and even adapting to completely new designs is a straightforward, efficient task. The addition of new cache fields, modification of coherency rules, insertion of a new cache-hierarchy level, and the introduction of new types of caches, can all be carried out in minutes and require no recompilation.

CacheLoader's powerful user request language provides verification engineers with the flexibility and controllability required to support a verification plan, especially when compared to the limited set of generation parameters supported by the original module. The simplicity in which new biasing options may be added further enhances the user's ability to target unique coverage goals. In contrast, enhancing the original reference tool to support a new biasing option requires careful consideration by the tool developer. Without the extra caution, the new function could lead to generation failures or loss of balance between existing options. Furthermore, the algorithm used by the reference tool is

not well-suited for cases that require the consideration of system-wide constraints.

7. CONCLUSIONS

This paper presents an ontology-based, CSP-powered cache-preloading technology. The tool's architecture follows principles of ontology based software to achieve complete separation between the cache loading engine and design dependent knowledge. Constraint satisfaction techniques are used to generate system initializations, satisfying the declarative modeling of design dependant rules, expert knowledge, and explicit user directives.

CacheLoader was tested with several IBM designs exhibiting different structures, coherency rules, and behavior. The tool demonstrated significant advantages over the traditional ad-hoc implementation, with a slight penalty of longer generation run-time. Adaptation to a new design is faster and is almost plug-and-play. The reusability level is higher, and domain knowledge is accumulated and shared between projects. This sharing of knowledge replicates the impact of experts in this unique area. The powerful user request language, and the simplicity of adding new fine grained biasing options, provides the flexibility and the control required to implement a verification plan and target unique coverage goals.

We foresee that other verification services will be built using similar principles, boosting the productivity of verification engineers and speeding up the construction of mature verification environments.

REFERENCES

[1] Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M. and Ziv, A. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. IEEE Design Test of Computers, Mar-Apr. 2004, 84-93

[2] Adir, A., Emek, R., Katz, Y., Koyfman, A.: DeepTrans - A Model-Based Approach to Functional Verification of Address Translation Mechanisms, MTV 2003

[3] Aloni, G. et al.: X-Gen: A Random Test-Case Generator for Systems and Socs, HLDVT 2002

[4] Barták, R., “Theory and Practice of Constraint Propagation”, CPDC2001, June 2001

[5] Bhatia, R. R., “Design and Evaluation of a Constraint Satisfaction Problem Solver-Based Cache Preloader” M.S.E.. Thesis, University of Texas at Austin, August 2008

[6] Bin, E. et al., “Using a Constraint Satisfaction Formulation and Solution Techniques for Random test Program Generation,” IBM Systems Journal, vol. 41, No. 3, 2002

[7] Dechter R., “Constraint Processing”. A book by Elsevier Science (USA) 2003

[8] Duineveld, A.; Stoter, R.; Weiden, M. R.; Kenepa, B.; and Benjamins,V. R.: "Wonder Tools? A Comparative Study of Ontological Engineering Tools", KAW 1999

[9] Ganzha, M. et al.: "Utilizing Semantic Web and Software Agents in a Travel Support System". In: A. F. Salam and Jason Stevens (eds.), Semantic Web Technologies and eBusiness, 2006, pp.325–359.

[10] Haque, F., Michelson, J. and Khan, K. The Art of Verification with Vera, Verification Central, 2001

[11] Iman, S., Step-by-Step Functional Verification with SystemVerilog and OVM., 2008. Hansen Brown

[12] Le H. Q., Starke W. J., Fields J. S., O'Connel F. P., Nguyen D. Q., Ronchetti B. J., Sauer W. M., Schwarz E. M., Vaden M. T. 2007: IBM POWER6™ Microarchitecture. IBM Journal of Research and Development, Vol.51, No. 6, Nov. 2007

[13] Minsky M.: “A Framework for Representing Knowledge”, MIT-AI Lab Memo 306, June, 1974

[14] Moskewicz, M. W., Madigan, C F.,Zhao, Y., Zhang, L., Malik, S., “Chaff: engineering an efficient SAT solver”, DAC 2001

[15] Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., and Shurek, G., "Constraint-Based Random Stimuli Generation for Hardware Verification", AI magazine, Vol. 28, pp. 13-30 2007

[16] Norton B., Cabral L., Nitzsche J.: ”Ontology-Based Translation of Business Process Models”. ICIW2009, pp.481-486

[17] Noy, N. F., Fergerson, R. W. and Musen, M. A.: "The Knowledge Model of Protege-2000: Combining Interoperability and Flexibility". EKAW 2000

[18] OMG, UML 2.0 Specification, http://www.omg.org/spec/UML/2.0, 2005

[19] Palnitkar, S., Design Verification with e, Prentice Hall, 2003

[20] Schrijver, A. Theory of Linear and Integer Programming. John Wiley & sons, 1998

[21] Stuecheli, J., “System Performance Scaling of IBM POWER6™ Based Servers” HotChips 19, Aug 2007