Software Cache Eviction Policy based on Stochastic Approach

Stoyan Garbatov and João Cachopo
Software Engineering Group
Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento, INESC-id
Lisbon, Portugal
stoyangarbatov@gmail.com and joao.cachopo@ist.utl.pt

***Abstract* — This work develops an innovative approach for guiding high-level software caches' eviction policy. The decision on which data to keep in the cache is made according to a stochastic analysis over the application data access behaviour. This approach shows it is possible to achieve high cache hit ratios with a reduced cache size. The effectiveness of the policy is tested and validated through the execution of two distinct benchmarks – the TPC-W and the oo7 benchmarks. The newly developed approach is flexible enough to be applied to any high-level software cache in an object-oriented system.**

Keywords-software cache; stochastic approach; performance; data access.

I. INTRODUCTION

A cache is a small, high-performance memory-buffer abstraction used to store temporarily data that is deemed to be important for whatever operations may be taking place currently or in the near future. Most of the time, the data held by the cache originates from a (much) larger and (several orders of magnitude) slower medium, which is either the source or provides storage for the whole range of existing data. As a result, caches provide increased system performance by offering shorter access times to data, keeping the available processing units busy with work. The most common restriction of a cache, however, is that it cannot hold all the existing data. This may happen for several reasons – the cache may be physically unable to provide enough storage space for all the available information, or, even if there is enough space, it may be better to keep the size of the cache to a minimum because a bigger volume of data (being held in cache) usually leads to slower execution times of the lookup operations.

The success of caching mechanisms results from the "principle of locality", which was first introduced by Denning [1]. The principle of locality, also known as locality of reference, has two basic variants, temporal and spatial. Over short periods of time, a program distributes its memory references non-uniformly over its address space, but the portions of the address space that are favoured remain largely the same for long periods of time. Temporal locality implies that the information that will be in use in the near future is likely to be already in use. Spatial locality states that the portions of the address space that are in use consist of a small number of individually contiguous segments of that address space. As a consequence, locality of space denotes that the referenced locations of the program in the near future are likely to be near the currently referenced locations.

Optimizing the design of a cache revolves around four aspects: maximizing the probability of finding a piece of data in the cache (the hit ratio), minimizing the access time to information already in the cache (access time), minimizing the delay due to a cache miss, and minimizing the overheads of cache management, such as propagating modifications to the means that backs the cache, or dealing with consistency protocols (cache coherence).

The principles upon which the concept of caching is based are present in many contexts and situations. This makes it possible to employ them in a variety of different contexts to improve the system performance. Caching mechanisms can be divided into two main categories, namely hardware caching and software caching. Given that the main purpose of this work is to improve the hit ratios of a high-level software cache, hardware caches are not considered here.

Significant research has been carried out in software caching. As has been pointed out, the four major characteristics upon which a cache can be improved are its hit ratio, access times, speed at which update propagations are performed, and coherence. If we group existing research according to affinity with these four aspects, a trend becomes apparent – namely, most cache-related work concentrates on coherence, as can be seen in [2], [3], and [4]. At the same time, the hit ratio is an important property, especially for software caches. It has been systematically identified as being the main reason leading to poorer performance of software cache approaches, in comparison with their hardware counterparts, as has been reported in [4] and [5].

Bennet et al. [6] identified and classified several classes of shared data accesses, in the context of distributed shared memory systems. They proposed a number of memory coherence approaches tailored for these access categories and demonstrated that specialized approaches can significantly outperform general ones, whenever the expected type of access behaviour manifests itself in a consistent fashion.

Dash and Demsky [7] presented an innovative distributed transactional memory system that mitigates the effects of network latency by prefetching and caching domain objects. The authors developed several extensions to the Java programming language with the goal of allowing the use of a distributed transactional memory within any application that employs their system.

The objective of this work is to develop an innovative stochastic approach for guiding high-level software caches. It consists in using a guided cache policy to decide which data to keep in memory and which data may be discarded. The cache policy is guided because it adapts to the behaviour displayed by the application, and its goal is to provide the highest possible cache hit ratio, while keeping in memory (cache) the minimum amount of data.

The article has the following structure. Section II describes the system. Section III presents the results obtained through the benchmark execution and evaluates the system effectiveness. Finally, Section IV derives the concluding remarks.

II. System Description

The system is composed of two parts: a stochastic access-prediction module and a high-level software cache. The access-prediction module is responsible for analysing the behaviour of the underlying application and in identifying the most common data access patterns performed. This information is subsequently used to guide the cache policy with the aim of improving its performance (at the level of its hit ratio). The software cache consists in a transparent data-storage component, responsible for supplying with data any request issued by the overlaying application, with the goal of improving the performance of an application.

A. Stochastic Behaviour Analysis

The stochastic behaviour analysis module is made up of three sub-modules: a code-injection module, a data-acquisition module, and a data-analysis module. An overview of their functionality is given here, while a detailed discussion of their implementation and behaviour may be found in [8], [9] and [10]. The model of Bayesian Updating, first presented in [8], is employed here for the stochastic behavioural analysis of the target application. An alternative model, based on discrete-time Markov Chains, may be seen in [9], whilst [10] deals with an Importance Analysis model.

The code-injection module is responsible for transforming the code of the target applications to inject the calls to the functionality present in the other modules. This code injection is performed in a completely automatic fashion by the system. It avoids the need for the application programmers themselves to perform any modifications whatsoever to their applications.

The data-acquisition module is responsible for acquiring behavioural data from the target application. This data describes how the application behaves, with regard to the data accesses that it performs. This module records which (application-domain) data is read and/or written, and in which contexts (methods, services, etc) this takes place.

Finally, the data-analysis module contains the implementation of the Bayesian Updating Inference model. This model corresponds to a stochastic approach for modelling the behaviour of the target application. The model uses as input the information collected by the data-acquisition module, about which domain data has been accessed by the application, and in which contexts. Initially, the input information is split into two sets of data. The first of these data sets is designated as *prior* and contains information about the target system behaviour observed in the past. The second set is called *current* and includes more-recent behavioural information. It covers the time period defined between the moment at which the prior set ends, to the current point in time. Once these two sets have been established, the Bayesian Inference model uses the current data to "update" the posterior, generating thus a third set, called *posterior*. The posterior set corresponds to the prediction generated by the model. It describes the expected behaviour of the application, in the near future, in terms of the domain data that it is going to access. This is presented in terms of the probabilities of reading and writing domain data, depending on the contexts through which the application passes during its execution.

B. Software Cache

This section describes the implementation of the high-level software cache and its policy. It should be noted that the term “high-level” is used here in the sense that the objects being cached correspond to actual domain object instances, rather than a derivate of an SQL result set or some other lower-level abstraction. The mapping from the format used by the underlying persistence layer to the domain object instances manipulated by the application is taken care by the fenix-framework (Fernandes and Cachopo [11]).

The software cache implements the identity map design pattern, Fowler [12]. This pattern prevents duplicate loading of objects from the persistence layer. Consequently, if a requested datum has already been loaded from the persistence layer, then the identity map returns the same instance of the already instantiated object. If it has not been loaded yet, then the object is retrieved and stored in the map, before being returned to the request that demanded it.

This cache is implemented on top of Java’s soft references. A normal Java reference, also known as a strong reference, guarantees that any object that is reachable through a chain of strong references is not eligible for garbage collection (GC). On the other hand, an object that is only weakly reachable is going to be discarded at the following cycle of the garbage collect. Soft references are not required to behave differently from weak references, but, in practice, softly-reachable objects are generally retained (in memory) provided that there is enough free space available to keep them there.

Continuing on to the implementation details, the high-level software cache keeps two collections into which it stores the loaded domain object instances. The first collection keeps soft references to all of its elements. This collection contains all the instances present in the cache. From the definition of a softly-referenced object, if the application does not hold a strong reference to them, the GC may discard them at will, if it deems it necessary to do so. However, they are usually kept in memory as long as it is not strictly necessary to evict them.

The second collection holds strong references to its elements, and guarantees that these can never be garbage collected, provided they remain in this collection. Objects being loaded into the cache are selectively added to the strongly-referenced collection. The main idea behind this collection is to keep only object instances considered to be important for the execution of the application, and that should be kept in memory even when they are not currently being used. The decision of adding an element to this collection belongs to the caching policy, implemented as follows.

The cache policy manager considers, on an instance-by-instance basis, if a given datum should be placed in the strongly-referenced collection. It employs the results generated by the stochastic behaviour analysis module to infer how strongly referenced these should be. The main criterion is to consider the access probability of the type of data (class) to which an instance belongs. If this probability exceeds a certain threshold, then the datum is deemed critical for the application operation and is inserted into the strongly-referenced collection, besides being added to the softly-referenced collection. This approach may be complemented to take into account further restrictions, such as the available free memory, space limitations that the software cache should not exceed, or proportions of different domain data types kept in memory, among others.

Additionally, due to the fact that the stochastic analysis model is dynamic, it reveals any behavioural change that may eventually come to pass within the target application. This would bring about an updating of the expected application domain data access probabilities. Furthermore, it would lead to a change in the data types considered critical by the cache policy manager, which would be reflected in the contents of the strongly-referenced collection, resulting, ultimately, in a caching policy that can adapt itself to deal adequately with any behavioural patterns the application may exhibit during its life cycle.

III. RESULTS AND EVALUATION OF THE SYSTEM

For the validation of the system presented in this paper, we used two distinct benchmarks. The first of these is the TPC-W benchmark, which was introduced originally by Smith [13]. This benchmark specifies an e-commerce workload that simulates the activities of a retail store website, where emulated users can browse and order products from the website. The main evaluation metric is the WIPS – web interactions per second that can be sustained by the system under test. The TPC-W benchmark execution is characterised by a series of input parameters that control its behaviour. Among these is the type of workload simulated by the benchmark emulated browser clients. The results presented in the article are associated with the "Shopping Mix" workload, which is composed of 80% read operations and 20% write operations. Regarding the main control parameters, they are as follows: number of emulated browsers - 10; ramp-up time - 600sec; measurement interval - 1200 sec; ramp-down time - 300sec; number of items in the database - 100 000; think time - 0.01 (this value ensures that the emulated browsers wait between 0.07 sec and 0.007 sec before making a new request to the server). The emulated browsers and the benchmark application server were run on the same physical machine.

The second benchmark is the oo7, firstly presented by Carey et al. [14]. This benchmark is often used to assess the performance of object-oriented persistence mechanisms. It strives to present a broad set of operations, allowing the building of a comprehensive performance profile. The oo7 benchmark was designed to boast properties common to different CAD/CAM/CASE applications, although in its details it does not model any specific application. A run of the benchmark executes a series of traversals, updates, and query operations over the underlying object model, and the performance metric used is the time that these operations take to execute.

The results obtained with our proposed approach to implement a cache policy shall be presented next. We omit a more thorough analysis of the correct behaviour and precision of the predictions of the stochastic behaviour analysis module, because this has already been performed in [8]. There, it is demonstrated that the module is capable of predicting with high precision the types of data being accessed by the target application in the contexts through which it passes during its execution. The term context can be defined to correspond to a procedure, service, or any other abstraction deemed appropriate to describe the scope within which the current operation is taking place. Moreover, the module is capable of predicting not only the type of data (domain classes) that is most likely to be accessed in a given situation, but also the effectively accessed object fields. For the discussion presented here, only access probabilities at the level of domain class shall be considered.

As has been explained in Section II.B, whenever a domain object instance is loaded into cache, it is always added to the softly-referenced collection. Additionally, the cache policy manager uses the stochastic access prediction module to determine what is the global (at the level of the whole application) access probability of the type of object being loaded. If the access probability exceeds a certain threshold, then the object instance is also added to the strongly-referenced collection, ensuring that it cannot be garbage collected. This policy shall be referred to as the DAP (data access pattern) policy for the reminder of this article.

Due to the fact that it is through the strongly-referenced collection that the cache policy effectively controls which objects are kept in memory longer than their actual use by the application, all cache hit and miss ratio results presented next are computed based on the contents of the strongly-referenced collection only.

For evaluating the effects of employing the DAP policy, the resulting cache hit ratios are compared with those obtained by the use of three alternative cache policies.

The first of the alternative policies decides whether to insert a given domain object in the strongly referenced collection as a function of a randomly generated number. The random generator employs a uniform distribution.

The second policy adds objects to the strongly referenced collection whenever they are first loaded by the application into the cache – the first objects to be loaded into the cache are the first to be made strongly reachable (this policy shall be henceforth referred to as "first load, first strongly referenced" – FLFSR).

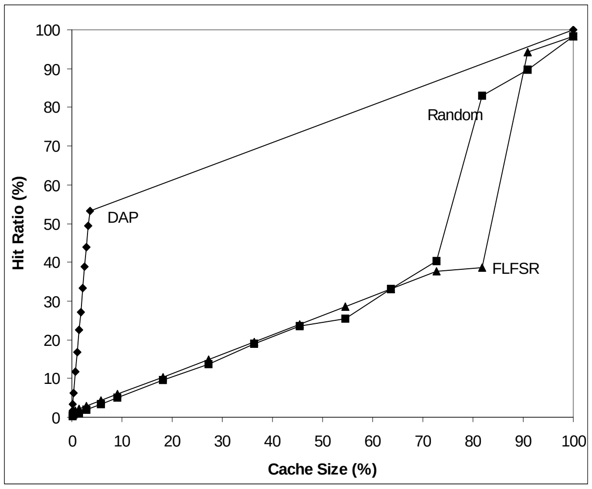

Figure 1. DAP, Random, and FLFSR policies - oo7

The third alternative corresponds to an LRU (least-recently used) policy. Its implementation is based on a synchronized and thread-safe version of the LRUMap structure of the Apache Commons Collections library. Because of the significantly different behaviour of an LRU policy, its comparison against the DAP policy shall be performed separately.

Furthermore, regardless of which policy is used, the strongly-referenced collection has an enforced maximum capacity. As such, for the Random and FLFSR policies, objects are inserted only if this capacity has not been reached. Generally, the dynamic nature of the DAP and LRU policies allows them to change the contents of the cache without exceeding the above threshold.

The results obtained from the execution of the oo7 benchmark for the DAP, Random, and FLFSR policies are shown in Fig. 1. The x-axis of the chart indicates the percentage of objects allowed to be strongly referenced in the cache, as a function of the total number of domain objects loaded into the cache during an execution of the benchmark. It has to be pointed out that due to the fact that both benchmarks access all of their domain objects during their operation, all of the existing persistent domain data ends up being accessed and cached during a single benchmark execution. The y-axis indicates the overall cache hit ratio achieved by a certain cache policy when the cache size is restricted to the value on the x-axis. Each of the dots presented in the graphs corresponds to the weighted average resulting from the measurements extracted from ten independent executions of a benchmark, for a given strongly-referenced cache size restriction.

As the results of the oo7 benchmark show, the DAP cache policy achieves better hit rate than both the Random and FLFSR cache policies, for the whole range of cache sizes. In particular, with only 3.6% of the total volume of domain data, the DAP policy achieves a hit rate of approximately 53%, whereas the Random and FLFSR policies require caching 76% and 84%, respectively, of the total volume of existing data to achieve a similar hit rate.

An interesting observation regarding the results from the DAP policy is that the data considered as important according to the stochastic analysis module (and thus suitable to be placed in the strongly referenced collection in the cache) corresponds to 3.6% of the total volume of existing domain data. This explains two peculiarities of the results observed for this policy. The first of these is the high cache hit ratio achieved for the relatively low volume of cached data (3.6%). It confirms the belief that the most frequently used data for a given application corresponds to a relatively small set of data. The second is the lack of measurements in the range of 3.7% to 99% along the x-axis. According to the behaviour prediction module, besides the 3.6% of data considered very important for the operation of the application, there is no other domain data that is even closely as likely to be needed by the application. Consequently, the cache policy cannot place any additional information in the strongly referenced part of the cache.

The results for the DAP (uninterrupted curves) and LRU (dotted curves) policies are shown in Fig. 2. The x-axis of the chart corresponds to a logic time scale, where a single unit corresponds to the realization of 10,000 lookup operations in the cache. The y-axis indicates the accumulated cache hit ratio up to a given point in the logical time scale. We chose a logic time scale instead of a real time scale because even though the benchmark is deterministic and performs all operations in the same order, differences in the execution time from one benchmark run to another would cause the different sampled curves to compress or expand with regards to one another, resulting into a rather deformed diagram. With a logical time scale, all curves are in synch with one another.

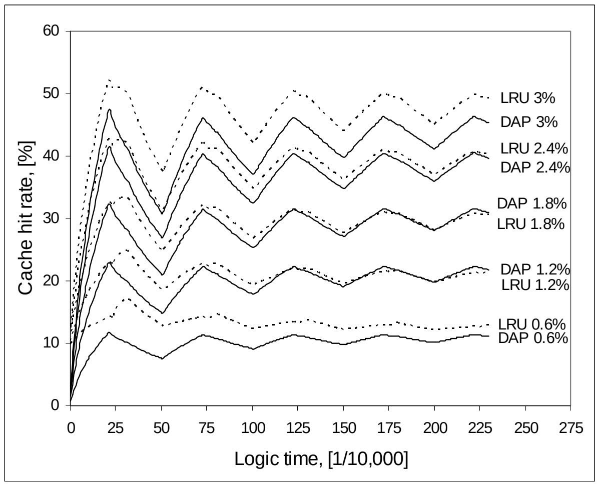

Figure 2. DAP and LRU policies - oo7

Fig. 2 presents five sample curves plotted for the DAP and LRU policies. Each of these is associated with a different cache size, corresponding to 0.6%, 1.2%, 1.8%, 2.4%, and 3% of the total volume of domain data. This relatively low percentage of domain data is due to the fact that only a very small part of the domain data is highly likely to be accessed in run-time. This domain data accounts for a maximum of 3% of all existing domain data. The results show that the LRU cache policy presents a better cache hit ratio for the great majority of cases,. The differences in average hit ratios between the two policies vary from 1.13% for 1.2% cache size up to 5.03% for 3% cache size, all in favour of the LRU policy. The weighted average of all measurements is 2.55% cache hit ratio difference in favour of the LRU policy. Regarding the effect on overall benchmark performance, there were no observable differences between the DAP and LRU policies.

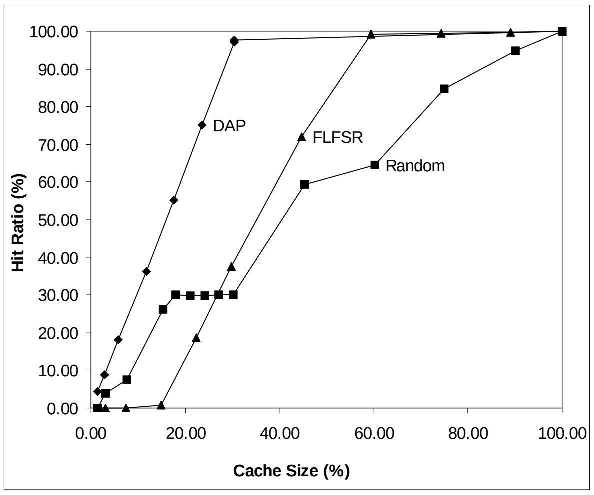

Figure 3. DAP, Random, and FLFSR policies - TPC-W

The results achieved from the execution of the TPC-W benchmark are discussed next. The hit ratio measurements for the DAP, Random and FLFSR cache policies can be seen in Fig. 3. The remarks to be made about these results are similar to the ones for the oo7 case, namely, the DAP cache policy presents cache hit ratios that are significantly better than the ones provided by the Random or FLFSR policies for any configuration.

Analysing the results of the DAP policy, we observe a practically linear growth in the hit ratio, starting from a hit ratio of 4.44% for cache size of 1.47% up to a hit ratio of 97.58% when the cache size corresponds to 30.42% of the domain data. The lack of measurements in the range of 30% to 100% of the cache size result from the same reasons presented for the oo7 benchmark – the domain data evaluated as important for the operation of the application corresponds to 30% of all of the existing domain objects; the remaining 70% of domain data are practically irrelevant, as they correspond to the remaining 2.42% of cache hit rate.

Considering the results for the uninformed caching policies, we are faced with a phenomenon not present in the oo7 benchmark results. This phenomenon consists in the existence of "plateaus" in the hit rate values achieved for a given range of cache sizes. For the Random policy, instances of this are the 30% hit rate in the range of 18% to 30% cache size and the 65% hit rate for 45% to 60% cache size. For the FLFSR policy, similar remarks are applicable to the 0% hit rate in the range of [0%, 15%] cache size and the 99% hit rate for the range of [59%, 100%] of cache size. These plateau phenomena may be explained by the caching of domain data that is practically irrelevant, from the point of view of the application needs. This leads to an increase in the volume of cached data without any significant increase in hit rate, which is what the plateaus effectively correspond to.

The final set of results, comparing the DAP and the LRU policies, are shown in Fig. 4. In this case, the x-axis corresponds to a real time scale where the unit corresponds to 20 seconds, whilst the y-axis indicates the accumulated cache hit ratio observed up to a given point in the benchmark execution. There are three curves for each of the two policies, corresponding to cache sizes of 9%, 12%, and 15%. For the TPC-W benchmark, the LRU policy displays an even more accentuated advantage over the DAP policy, with regards to the hit ratio they achieve. In terms of differences between average hit ratios, the LRU policy leads with 9.2% for the 9% cache size, 13.8% for the 12% size, and 9.8% for the 15% size. This leads to an overall average hit ratio advantage of 10.9% in favour of the LRU policy. Yet, even though the average values give a clear advantage to the LRU policy, the observed behaviour for the LRU hit ratio is rather irregular, at least when compared to that of the DAP policy, whose results are very close to flat horizontal lines.

The most-significant difference between the two approaches in the case of the TPC-W benchmark (unlike what was seen for the oo7 benchmark) is the performance variations observed between the versions running with the DAP and the LRU policy. These variations are due only to the performance of the policy itself, rather than, for example, to the contents of the cache, because the contents of the real cache, which dictates the overall benchmark performance, is the same for both versions. Only the contents of the strongly-referenced collections are distinct and it is against those that the hit ratios are measured.

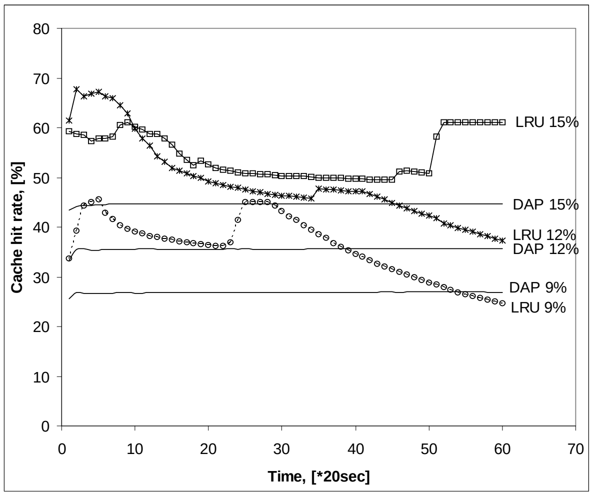

Figure 4. Using DAP and LRU policies with TPC-W

A comparison of the benchmark's performance when using the DAP and the LRU policies is shown in Fig. 5. The x-axis indicates the number of emulated browsers (EBs) employed for a given benchmark run and the y-axis shows the relative throughput gain achieved for a given number of EBs. The values shown correspond to the speedup relative to executing the benchmark version with LRU cache policy with 1 EB. These results show that both versions perform approximately the same amount of work for 2 EBs, which is approximately 100% more than the LRU version with 1 EB. However, as the number of EB increases, the results show a growing discrepancy in the benchmark performance between the LRU and the DAP policy versions. For four EBs, the LRU benchmark version performs approximately 210% more work than the baseline, whereas the DAP version manages over 310%. This difference is even more accentuated for 10 EBs, where the performance of the benchmark with the LRU policy has remained practically the same as the one from the 4 EBs configuration, whilst the DAP version has grown up to over 560%.

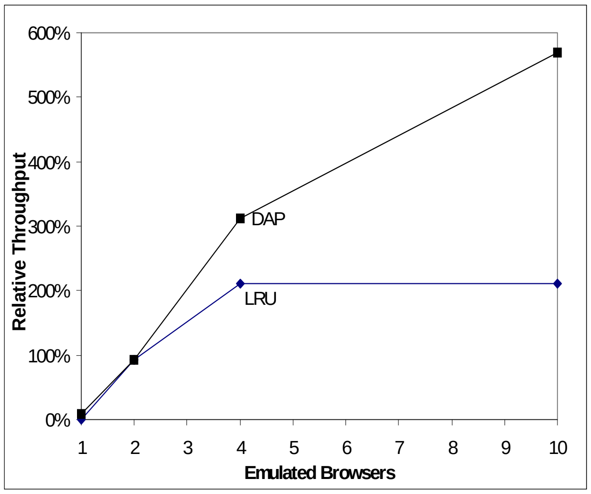

Figure 5. DAP and LRU throughput comparison, TPC-W

The most reasonable explanation for this phenomenon is that the synchronization present in the LRU policy implementation causes a bottleneck in multithreaded scenarios, leading to the poor performance gains observed in the results. Assuming this is the case, then the DAP policy would be the preferred alternative for situations where multithreading is common, while the LRU would be more appropriate for single threaded configurations.

IV. CONCLUSIONS

This paper presented a new approach for guiding the cache policy of a high-level software cache. This new approach employs a stochastic analysis based on Bayesian Updating Inference, which is responsible for predicting the behaviour of the target application, regarding its domain data needs. Based on the generated predictions, the cache policy is capable of deciding which domain objects are to be cached, leading to high cache hit rates with relatively low volumes of cached domain data.

The effectiveness of this approach was tested with two very different benchmarks – the TPC-W and the oo7 – by comparing it against three different cache policies. The results illustrate the usefulness of employing dynamic adaptive approaches for guiding high-level software caches, by taking into consideration the behaviour of the target application.

ACKNOWLEDGMENT

This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the Specific Targeted Research Project (STReP) Cloud-TM, which is co-financed by the European Commission through the contract no. 257784. The first author has been funded by the Portuguese FCT (Fundação para a Ciência e a Tecnologia) under contract SFRH/BD/64379/2009.

REFERENCES

[1] Denning, P. J. and Schwartz, S. C., 1972, Properties of the working-set model, Commun. ACM, 15, (3), pp. 191-198.

[2] Sandhu, H. S., Gamsa, B. and Zhou, S., 1993, The shared regions approach to software cache coherence on multiprocessors, SIGPLAN Not., 28, (7), pp. 229-238.

[3] Lilja, D. J. and Yew, P.-C., 1991, Combining hardware and software cache coherence strategies, Proceedings of the 5th international conference on Supercomputing, Cologne, West Germany, ACM, pp. 274-283.

[4] Adve, S., Adve, V., Hill, M. and Vernon, M., 1991, Comparison of hardware and software cache coherence schemes, SIGARCH Comput. Archit. News, 19, (3), pp. 298-308.

[5] Chen, T., Zhang, T., Sura, Z. and Tallada, M., 2008, Prefetching irregular references for software cache on cell, Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, Boston, MA, USA, ACM, pp. 155-164.

[6] Bennett, J., Carter, J. and Zwaenepoel, W., 1990, Adaptive software cache management for distributed shared memory architectures, 17th Annual International Symposium on Computer Architecture, pp. 125-134.

[7] Dash, A. and Demsky, B., Integrating Caching and Prefetching Mechanisms in a Distributed Transactional Memory, To Appear in IEEE Transactions on Parallel and Distributed Systems.

[8] Garbatov, S., Cachopo, J. and Pereira, J., 2009, Data Access Pattern Analysis based on Bayesian Updating, Proceedings of the First Symposium of Informatics (INForum 2009), Lisbon, Paper 23.

[9] Garbatov, S. and Cachopo, J., 2010, Predicting Data Access Patterns in Object-Oriented Applications Based on Markov Chains, Proceedings of the Fifth International Conference on Software Engineering Advances (ICSEA 2010), Nice, France, pp. 465-470.

[10] Garbatov, S. and Cachopo, J., 2010, Importance Analysis for Predicting Data Access Behaviour in Object-Oriented Applications, Computer Science and Technologies, 1, pp. 37-43.

[11] Fernandes, S. and Cachopo, J. A New Architecture for Enterprise Applications with Strong Transactional Semantics. Lisbon: INESC-ID / IST, 2011.

[12] Fowler, M., 2003, Patterns of enterprise application architecture: Addison-Wesley Professional.

[13] Smith, W. TPC-W: Benchmarking An Ecommerce Solution. Intel Corporation, 2000.

[14] Carey, M., Dewitt, D. and Naughton, J., 1993, The OO7 benchmark, ACM SIGMOD International Conference on Management of Data, pp. 12-21.