Journal of Software Engineering and Applications, 2020, 13, 191-205
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Evaluation of an Evolutionary Algorithm to Dynamically Alter Partition Sizes in Web Caching Systems

Richard Hurley, Graeme Young

Department of Computer Science, Trent University, Peterborough, ON, Canada
Email: rhurley@trentu.ca

How to cite this paper: Hurley, R. and Young, G. (2020) Evaluation of an Evolutionary Algorithm to Dynamically Alter Partition Sizes in Web Caching Systems. *Journal of Software Engineering and Applications*, **13**, 191-205.
https://doi.org/10.4236/jsea.2020.139013

Received: June 29, 2020
Accepted: September 26, 2020
Published: September 29, 2020

Abstract

There has been an explosion in the volume of data that is being accessed from the Internet. As a result, the risk of a Web server being inundated with requests is ever-present. One approach to reducing the performance degradation that potentially comes from Web server overloading is to employ Web caching where data content is replicated in multiple locations. In this paper, we investigate the use of evolutionary algorithms to dynamically alter partition size in Web caches. We use established modeling techniques to compare the performance of our evolutionary algorithm to that found in statically-partitioned systems. Our results indicate that utilizing an evolutionary algorithm to dynamically alter partition sizes can lead to performance improvements especially in environments where the relative size of large to small pages is high.

Keywords

Evolutionary Algorithm, Web Cache, Partition, Simulation, Performance Analysis, Hit Rate

1. Introduction

Today's Internet is a vast network of interconnected computing devices, through which information is shared at an extremely high volume and speed. Over time, the amount of data transferred between computing devices has increased dramatically. To put it into perspective, the modern Internet consists of an environment of video streaming, online television, massively multiplayer online games, and live music streaming. Contrast this to an early Internet of utilitarian, sparse Web pages featuring little more than a few paragraphs of plain text and perhaps several small images. When juxtaposing that environment with the aforementioned modern one, the increased demand on network infrastructure becomes obvious. Although technology has attempted to keep pace with the increase in demand, the risk of a Web server being inundated with more requests than it can handle is an ever-present one. The amount of data circulated through the Internet has been doubling every six months [1].

Not only is there a huge volume of information shared through the Internet, the information tends to vary quite significantly based on the amount necessary to transmit [2]. Atypical Web page could consist of text, image, sound, and video files (or any combination of the aforementioned objects) all of which are of potentially different sizes. For this work, Webpages will be simplified and divided into two categories: *small* pages and *large* pages.

With the sharing or transferring of objects between computing devices comes the issue of managing and mitigating delays that may arise from physical distance, network loads, and/or the amount of information transmitted [3] [4]. While Web caching, the process of storing and managing Web pages in multiple locations, does not directly address the issue of page size, it can however address the issue of multiple concurrent accesses.

A typical Internet architecture consists of groups of computing devices sharing large volumes of information at a high speed. In its most basic form, the environment can be thought of being comprised of two classes of devices: servers (provide the information), and clients (request the information) [5]. When a client requests a Web page, the server relays a copy of that page to the client and then waits for another request. However concise this approach may be in theory it does not reflect the complex reality of the Internet with multiple clients accessing a server at any given time, and servers containing multiple pages that may be requested. A direct result of this is that as the number of requests to a server increases, the server takes longer to respond to each individual request due to the increased load (number of requests in a given amount of time). Web caching has been used in these networks to alleviate the load placed on servers by reducing the number of requests actually being processed by the server [6] [7] [8].

Physically, a Web cache is a storage medium that contains copies of Web pages (or objects) from a Web server, with page content determined, at least in part, by what pages are requested by clients. By storing copies of Web pages requested by clients, a Web cache can process future requests for those particular pages without involving the actual server. This is based on the assumption that those cached pages will be requested again at some point in the immediate to near future by other clients [7]. Web caches may be located in the client itself, in a proxy server, or in a physically separate device somewhere on the Internet [9]. In addition to physical location, it is also possible that several distributed caches, each one of which contains copies of Web pages from the server.

Previous research has shown that partitioning Web caches has led to performance improvements over systems that do not partition [10] [11]. Partitioning is a technique where a Webcache is divided into two fixed-sized partitions which are then allocated for specific-sized Webpages (e.g., one could have a partition for large pages and a partition for small pages). Much of the work done on partitioning Web caches has been limited to static partitioning where the size of the partitions remained fixed. In this paper, we examine dynamic partitioning where the size of a partition for a particular Web page type can be increased or decreased based on the prevalence of that type of page. The approach we will use to adjust the partition sizes will be based on an evolutionary algorithm [1] [12] [13] that attempts to optimize an objective function.

This paper will be organized as follows: Section 2 focuses on the basic model, with discussion provided on the choice of parameter. In Section 3, we will give a detailed outline of the particular evolutionary algorithm employed in our study. Section 4 presents some of the experimental results generated to investigate the properties of our model, and its advantages/disadvantages compared to existing models. Finally, in Section 5, we will provide a summary of findings and suggestions for future research.

2. Model Description

Our Web caching system model is divided into two parts: a Web page request reference model and a Web cache model. In this paper, we are concerned with the relative performance between a dynamic partitioning Web cache system and one which uses static partitioning. Since we are not concerned with the absolute performance of the system, we can make some simplifying assumptions with our system models.

2.1. Web Page Request Reference Model

The pages and objects stored in a Web cache and their request probabilities vary over time. Pages such as a news article, viral videos, course assignments, memes, etc. become popular for periods of time and then eventually are accessed less frequently. To capture this behavior, we use a variant of the dynamic page reference model described in [14].

2.1.1. Page Popularity

The page reference model for a system with M Web pages is shown in **Figure 1**. This model assumes that a Web page can be in one of two states: normal and popular. Web pages in the popular state are v time more likely of being requested than a page in the normal state.

The model also assumes that there are two types of pages: *conventional* and *potentially popular*. Conventional pages remain in the normal state while potentially popular pages alternate between the states based on an underlying Markov chain. The rate at which a page transitions from a normal to popular state is λ_1 and from popular to normal is λ_2 with the time spent in either state is assumed to be exponentially distributed. Finally, we let $M_0 < M$ denote the number of potentially popular pages that are present in the system. With this type of model,

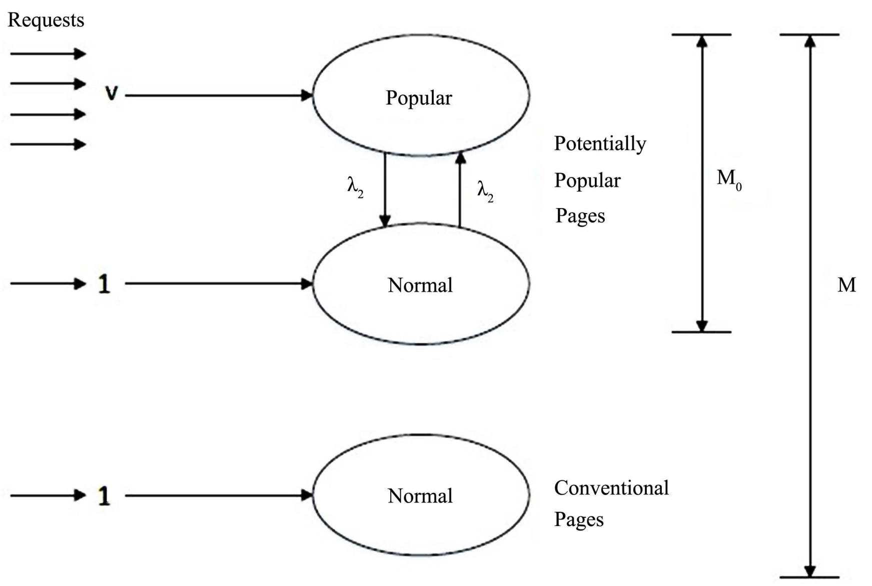

Figure 1. Dynamic page reference model.

the model is able to generate page requests with high coefficients of variation, an attribute that has historically shown to be desirable in such systems [15].

2.1.2. Page Size

To simplify our Web page reference model, we assume that there are only two different sizes of Web pages, *large* or *small* (with a large page being k times the size of a small page). Small pages have a service time (time to retrieve a copy of the page from the Web server) that is assumed to be exponentially distributed with a mean rate of μ^{-1}, and large pages have a exponentially distributed service time of $k\mu^{-1}$.

With the increased presence of images and videos, Web pages are increasing in size, however, the majority of Web pages are still relatively small (less than 1000 KB) [16]. As a result, we assume that the probability of requesting a small page p is 0:9 making the probability of requesting large pages $1 - p$.

2.2. Web Cache Model

Our Web cache model is comprised of a page replacement model, an architectural model, and a storage mode1.

2.2.1. Page Replacement Model

One of the main components of a caching system is the page replacement algorithm which is responsible for discarding pages in the Web cache once it becomes full to make room for new pages. Although there are several different page replacement algorithms, we use Least Recently Used (LRU) [17] which selects the least-recently used page (determined from the last accessed timestamp) to be removed. Since we are concerned with relative performance, using the same replacement algorithm for our systems under investigation will not create any unfairness.

2.2.2. Architectural Model

Our work expands on a Web cache model that was introduced by [18], and is shown in Figure 2. The Web caching system model has a finite population of N clients interacting a Web cache which is partitioned into a small page cache partition (SPCP), and a large page cache partition (LPCP).

Page requests are generated by the clients after an exponential think time (z) and sent to the Web cache. When the Web cache receives a page request, the cache first checks if there is a copy stored locally. If the page is found, it is simply returned to the client. However, if no copy of the requested page can be found, the request will be sent to the originating Web server, a copy is made at the Web cache, and the Web page is returned to the client. If the appropriate partition for the page size of the request is full at the Web cache, LRU is used to select a page for removal.

It is assumed that if the request cannot be satisfied by the Web cache and must be retrieved from the originating Web server, the processing time is to be μ^{-1} (this includes the service time and propagation delay). If the request can be satisfied by the Web cache, then the processing time is assumed to be $0.5\mu^{-1}$. If the request is for a large Web page, all the times are assumed to be k times larger.

2.2.3. Storage Model

We consider two variations of the cache storage model: a statically partitioned cache and a dynamically partitioned cache. The key difference is that the statically partitioned cache assumes that the partition sizes (measured in terms of the number of small pages that can be stored) remain fixed throughout the duration of the simulation while the partition sizes in the dynamically partitioned caching system vary according to an evolutionary algorithm. For either case, a partitioned cache in general treats large and small pages differently. From Figure 2, it can be seen that the cache is split into two separate areas: one for large pages and one for small pages. It is assumed that the ratio of space reserved for large pages is (P_L). Whenever a small page is brought into its partition, the available amount

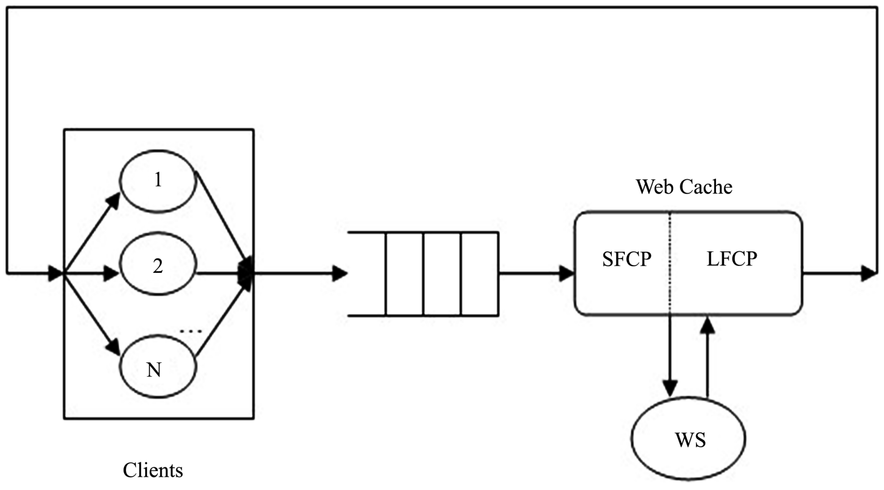

Figure 2. Web caching architecture model.

of storage in that partition is decreased by1. Similarly, when a large page is brought into its partition, the available amount of memory in that partition is decreased by k (the reverse occurs when a page is removed from its partition). If a partition is full, and the size of the partition cannot be adjusted through the evolutionary algorithm, then the LRU (Least Recently Used) algorithm is used to free enough space for that page to then be cached.

3. Evolutionary Algorithm

Evolutionary algorithms, as the name implies, are a class of algorithms for solving complex problems using processes that mimic those found in nature, specifically the processes of evolution [12]. The natural processes replicated in evolutionary algorithms include: mutation, breeding, natural selection, and evolution. In our research, we apply evolutionary programming which uses selection, controlled through a tournament, to determine candidate elements for successive generation [19] (see **Figure 3**). Each trial solution in the Web cache population faces competition against a preselected number of opponents and receives a win if it is at least as good as the competition. The selection process eliminates those elements with the least number of wins (much like *survival of the fittest*).

Evolutionary programming makes use of mutation to invoke variety in the population. The mutation operation simply changes aspects of the population according to a statistical distribution, with the severity of the mutations being reduced as the global optimum is approached. In a dynamic environment such as a Web cache system, a global optimum may not be possible since the behavior of the system constantly changes.

Of particular interest in this research is to examine an evolutionary algorithm that is dynamic in nature: that is, develop an algorithm that can perform adjustments while the system is in operation (in our case, at the point when pages are required to be stored in a Webcache). Such predictive systems are well suited to be used in an Internet environment where Web pages and object can over time and as well, users change habits change [12].

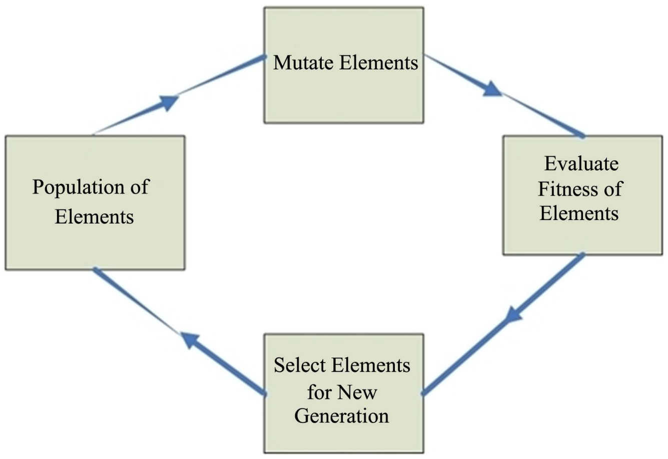
Mutate Elements
Evaluate Fitness of Elements
Select Elements for New Generation
Population of Elements

Figure 3. Evolutionary programming cycle.

One issue that arises from the use of an evolutionary algorithm in the context of a dynamic system is the determination of an optimum solution. In a relatively static environment, a fixed optimum solution can usually be determined since the underlying factors do not change significantly. However, in a dynamic Web environment the underlying factors change constantly and as a result, the optimum solution changes as well, making a *fixed best* solution impossible [12].

In order for an evolutionary algorithm to function, parameters must be provided that can be altered, or *mutated*, as required. In our model, these consists of:

- *SPStart*: defined to be the starting size of small page partition (must be less than or equal to maximum cache size and greater than or equal to 1).
- *SPAltAmt*: defined to be the amount to change small page partition size by *maxChange* units during any single mutation. For our system, we choose *maxChange*to be 4 as this provides a reasonable balance between the time to converge on a solution and the accuracy of solutions provided.
- *SPMin*: minimum size of small page partition (must be less than or equal to maximum cache size and greater than or equal to 1).
- *LPStart*: starting size of large page partition (this is set to whatever is left over from maximum cache size once small partition start size has been determined).
- *LPMin*: minimum size of large page partition (this is set to whatever is left over from maximum cache size once minimum small partition size has been determined).

When a change of *maxChange* units is to be applied, the actual amount of change is determined based on a uniform distribution between 1 and *maxChange* to avoid introducing a bias into the system. This is fundamental to ensuring reliable results from the evolutionary algorithm, as the amount of the mutation must be random to ensure that solution spaces are searched in the most effective way possible.

The algorithm used for implementing the evolutionary approach is as follows (for a more detailed presentation of the algorithm, please see [13]):

1) A panel of judges is created (a best solution to which candidates in the current genetic pool are compared). The decision as to how many judges to include is based on the number of candidates in the genome pool (the collection of solutions that will be compared to the judges). A value too large will result in an excessive amount of time to converge on a solution. Too small a number could result in the algorithm prematurely converging on a solution. For our algorithm, we use two judges and a genome pool of five.

a) On the first iteration, to get the algorithm started, random solutions are provided to the judges with *SPMin*, *LPMin*, *SPStart*, and *LPStart* being set to uniform random values between 0% and 50% of the total cache size (this allows free space in the cache to be reallocated from one partition to another by the evolutionary algorithm).

b) A simulation run is performed for solutions represented by the judges to gather statistics that will then be compared to those generated by candidate solutions in the next step.

c) Five candidate genomes (*i.e.*, trial solutions) are generated with *SPStart*, LPStart, *SPMin*, and *LPMin* initially set to uniform random values between 0% and 50%. Each of these candidates is measured against the current judges to determine fitness (*i.e.*, if a candidate exhibits better performance than one of the judge, the candidate replaces the judge in the next generation). Fitness is determined based on performance measures such as cache hit rate.

2) On each successive generation, for each candidate in the pool:

a) A new cache partition assignment is generated, using the traits encoded in the current candidate genome (*SPMin*, *LPMin*, *SPStart*, *LPStart*).

b) These traits are mutated in the candidates by randomly determining for each of *SPM in* and *SPStart*, if the mutation will be positive or negative, and the amount of the mutation ($1 \le amtChange \le maxChange$).

c) A simulation run is initiated using the new cache partition assignment.

d) Fitness results for the current candidate are then compared to that of the judges to see if any of the judges should be replaced.

3) Step 2 is then repeated (with the performance measures for each of the candidates and judges reset before the next generation) until there are five consecutive generations that show no change in the status of the judges. At this point, we assume we have the best obtainable solution for cache partition assignments in the best performing judge.

4. Results

The results from this study were generated using a discrete event simulation. The focus of the work is on applying an evolutionary algorithm to a Web caching problem and determining its ability to explore a large solution space in an efficient manner. We did examine the impact that the parameters of our evolutionary algorithm had on the performance of system and found that varying the number of judges, number of candidates, ratio of the number of candidates to number of judge, and mutation degree did not have a significant impact on system performance (other than to vary the number of generations required to achieve convergence).For a more detailed examination of these experiments, please see [13]).

In this paper, we will examine the impact of the cache size (C), relative size of large to small pages (k), percentage of small pages (p), number of Web pages (M), and the relative performance benefits of a Web caching system that utilizes dynamically-controlled partition sizes against to a system that used statically-assigned partition sizes. The performance measures of interest for our study is the hit rate of the Web caches. To reduce the number of experiments, we examine the system under three different loading scenarios: low system utilization (server utilization ≈ 50%), medium system utilization (server utilization ≈ 70%), and high system utilization (server utilization ≈ 90%).

We begin with Figure 4 which examines which examines the effects that varying the relative size of large to small pages (*k*) has on the hit rate for various cache sizes in a system under a medium load(the trends for low and high loads were similar and not shown in this paper).

The results indicate that varying the relative size of large to small pages (*k*) does not have a significant impact on the performance of our dynamically partitioned Web caching system. The most likely reason for this is that the percentage of small pages in the Web cache is initially assumed to be high and so increases in hit rate come mainly from the expansion of the small page partition. Thus, the actual net effect of changes to storage space (through alterations to *k*) for large pages would not have a significant impact on the resulting hit rate. Figure 4 shows that the maximum decrease we see in hit rate occurs when the cache size is 35%, and results in a decreases of only 3.9% as *k* is increased from 2 to 10. The graph does indicate, however, that increasing the percentage of Web pages cached does lead to an increase in hit rate as one would expect (up to 43% in model).

We believe that the dynamic nature of partition sizes achieved by our evolutionary model may provide some measure of stability as *k* fluctuates, at least with respect to the performance measure of hit rate. Further evidence of this can be observed in Figure 5, where we investigate the effect on hit rate of altering the percentage of small pages (*p*) for various values of *k* in a medium load system. From Figure 5, we can observe that the hit rate, while varying the probability of a page being small (a larger value of p would lead to a higher hit rate as more Web pages are small), does not change significantly based on k. This helps to confirm the theory that our evolutionary algorithm focuses on hit rate, which we can observed from the next graph (Figure 6), works to the detriment of large page-related metrics such as byte hit rate.

Byte hit rate measures the number of bytes satisfied by the cache, in relation to the total number of bytes requested. A larger value of *k* indicates that the relative size of a large page increases resulting in a decrease in the byte hit rate.

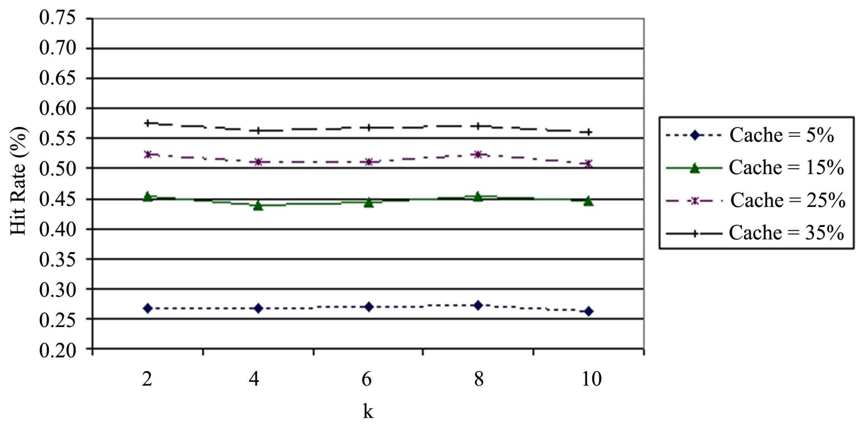

Figure 4. Effects of relative size of large pages (k) on hit rate (medium load) for various cache sizes (C), $\mu = 1$, $v = 10$, $z = 100$, p = 0.9, $M_0 = 10\%$, $M = 1000$.

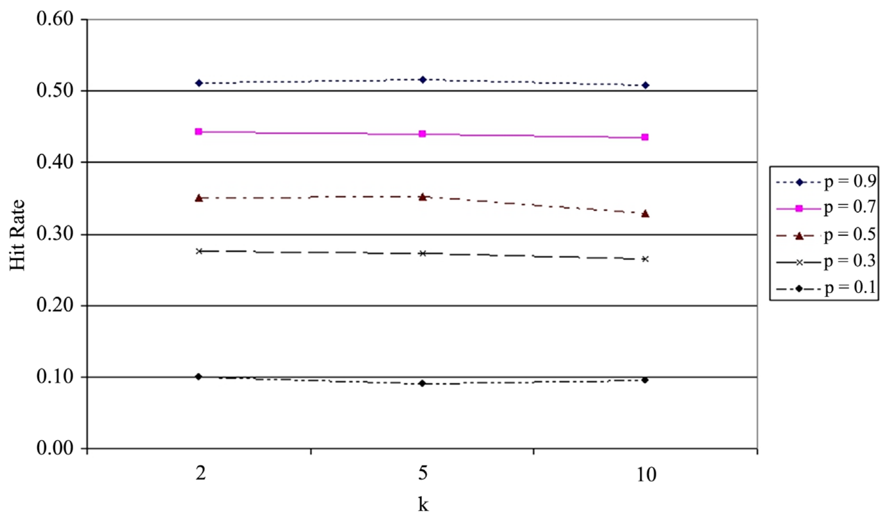

Figure 5. Effects of relative size of large pages (k) on Hit Rate (medium load) for various percentages of large pages (p), $\mu = 1$, $v = 10$, $z = 100$, $C = 25\%$, $M_0 = 10\%$, $M = 1000$.

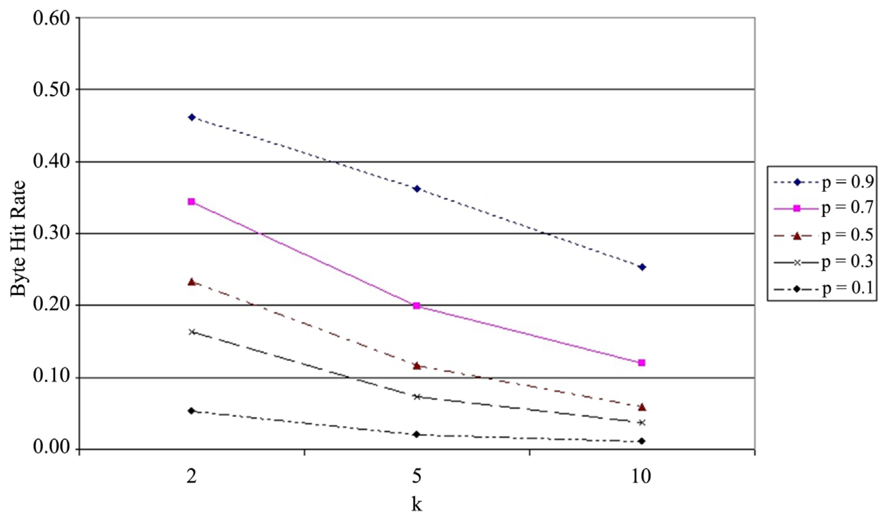

Figure 6. Effects of relative size of large pages (k) on byte hit rate (medium load) for various percentages of large pages (p), $\mu = 1$, $v = 10$, $z = 100$, $C = 25\%$, $M_0 = 10\%$, $M = 1000$.

Similar trends were observed in systems with low and high loads and therefore not presented here.

We now examine the effects that the number of Web pages has on the performance of the system. In Figure 7, we show the impact that increasing the number of Web pages has on hit rate for select values of the relative size of large pages (k) in a medium load system. From Figure 7, we can see that the number of Web pages present used in the model does not significantly alter the behavior of the system: the hit rate experiences an increase of about 3.6% as the number of pages (M) is increased. We can also see that the value of the relative size of large to small pages (k) does not seem to impact the hit rate as the number of pages increases(for byte hit rate, larger values of k would lead to a lower value as fewer pages could be kept in the cache [13]). This allows us to run our model

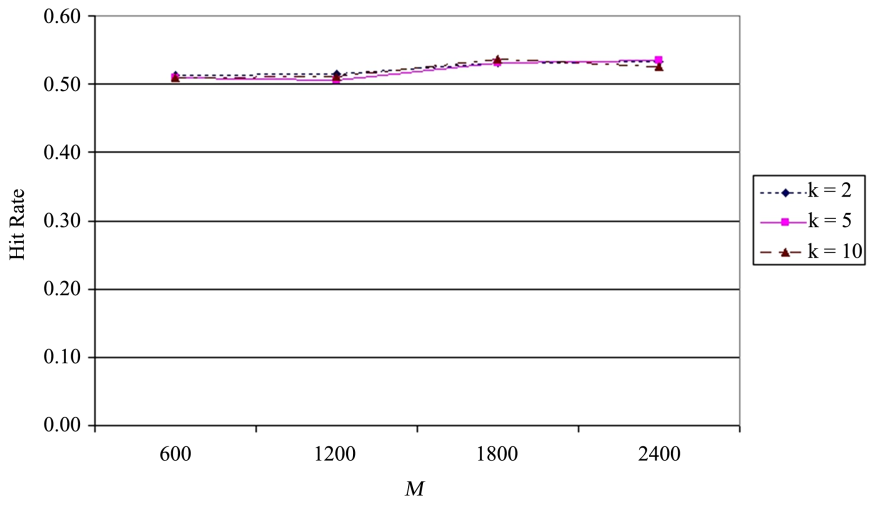

Figure 7. Effects of the number of web pages (M) on hit rate (medium load) for various percentages of large pages (p), $\mu = 1$, $v = 10$, $z = 100$, $C = 25\%$, $p = 0.9$, $M_0 = 10\%$.

with a reduced number of Web pages and be able to extrapolate the trends to larger systems while lowering the amount of time necessary to generate results.

Lastly, we compare the performance of our dynamically-partitioned Web caching system to statically-partitioned systems (the partition sizes for the large and small pages remain fixed for the duration of the simulation). Also, for interest, we will include the results those of a single-partition cache (large and small Web pages share the same cache). The purpose of these experiments is to gauge the effectiveness of the evolutionary aspect of our model, relative to a system which does not employ such a strategy. We compare the results for the three loading scenarios (low, medium and high) in an environment where the relative size of a large to small pages (k) is 2 and 10. The results are shown in **Figure 8** and **Figure 9**.

As can be observed in **Figure 8** and **Figure 9**, the resulting hit rates differ by a significant amount between the various static solutions, the evolutionary algorithm solution, and the single-partition solution. We can see from **Figure 8** that the single-partition cache system tends to outperform our evolutionary algorithm when the ratio of large to small pages is low (k = 2). When partitioning is involved, our dynamically-partitioned system outperforms the randomly-chosen static solutions. We believe that this shows the benefits of altering partition size while the system is operating. It is also interesting to note that there little variation in hit rates as the load on the systems goes from low to high indicating that Web caching in general is not affected by the amount of user traffic.

When we increase k to 10, the results become more beneficial for our evolutionary algorithm approach. As shown in **Figure 9**, we can see that our dynamically-partitioned system consistently provides an improvement in hit rate of up to 4.8% beyond that shown by the single-partition cache. This is most likely due to the fact that the changing of partition sizes accommodate numbers of pages more effectively as page sizes are increased.

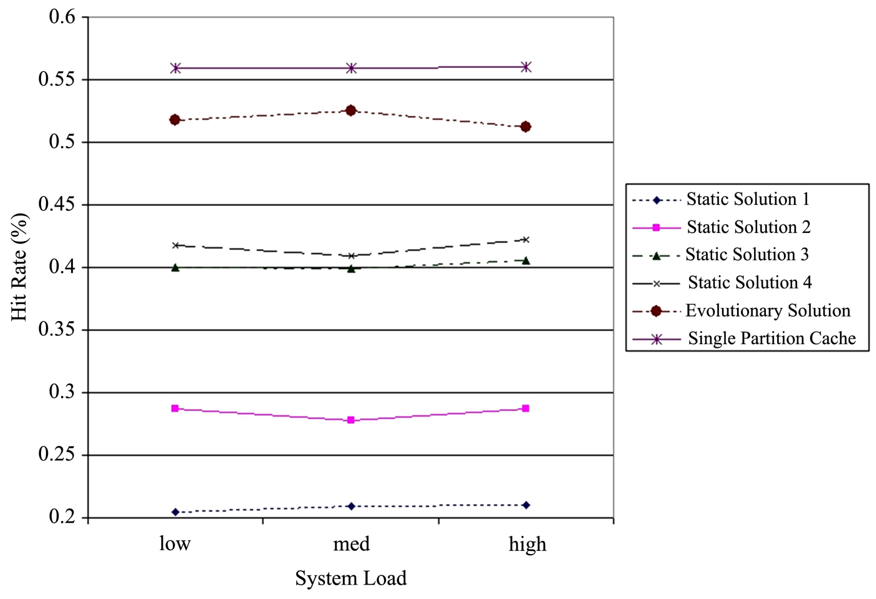

Figure 8. Hit rates of evolutionary algorithm approach versus statically-partitioned (k = 2), $\mu = 1$, $v = 10$, $z = 100$, $C = 25\%$, $p = 0.9$, $M_0 = 10\%$, $M = 1000$.

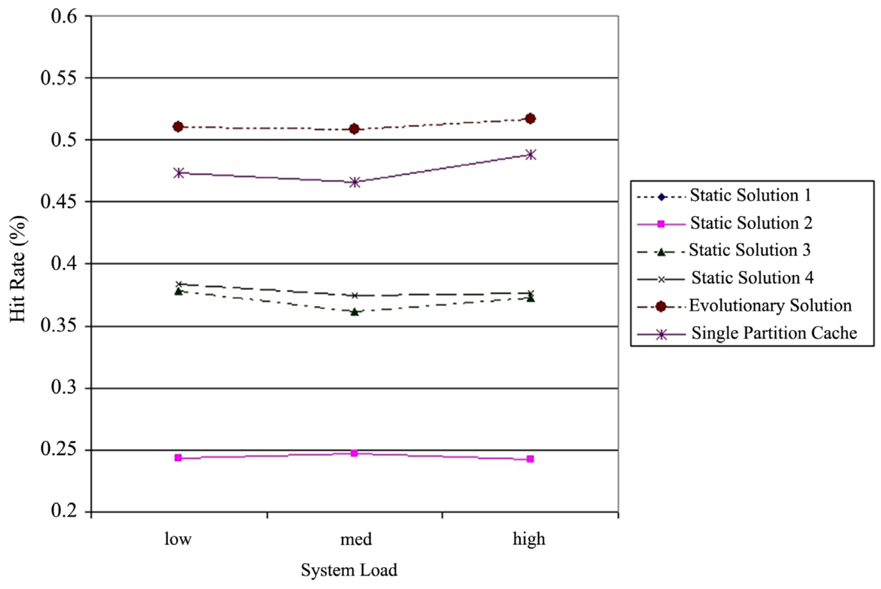

Figure 9. Hit rates of evolutionary algorithm approach versus statically-partitioned (k = 2), $\mu = 1$, $v = 10$, $z = 100$, $C = 25\%$, $p = 0.9$, $M_0 = 10\%$, $M = 1000$.

5. Conclusions

The selection of the partition size is an important decision in a Web caching system [11]. In this paper, we have investigated the use of an evolutionary algorithm to dynamically control partition size in a Web cache system. We have established that from our model, utilizing an evolutionary algorithm seems to outperform a statically-partitioned Web cache, and that the performance improvement tends to increase with the relative size of large to small pages.

We analyzed our system under multiple loading scenarios, different values of relative size of large to small pages, increasing ratios of larger pages, and various choices for the total number of Web pages. Our research showed that in terms of hit rate, our evolutionary algorithm appears to cope well with increases in the relative size of large to small pages. Byte hit rate, on the other hand, suffered from a decrease as the relative size of large to small pages was increased, which indicates that the caching strategy employed by our model tends to favor hit rate over byte hit rate. It also speaks to the idea that the large page partition becomes a limiting factor to performance as relative size of large to small pages is increased. As for changes to the total number of pages, we found that system behavior remained relatively constant, regardless of the number of Web pages. This demonstrates that the number of Web pages is not a significant factor in algorithm behavior.

Our last set of experiments compared a dynamically-partitioned system (using an evolutionary algorithm) with several statically-partition systems and a system without partitioning. We found that the evolutionary algorithm solution outperforms the statically-partitioned systems and has the potential to outperform a traditional Web cache in terms when the relative size of large to small pages increases. This finding is significant as the choice of partition size does affect performance and thus, a difficult parameter to determine. Our algorithm attempts to adjust this value as conditions vary.

Future research in this area could include investigating the mechanisms of the evolutionary algorithm such as allowing the mutation amount to become variable and providing alternative methods of comparison between judges and candidates to lead to further gains in performance. Data mining of cache logs could also be investigated to assist the evolutionary algorithm in its initial partition assignment in an attempt to arrive at the "optimal" solution more efficiently. Another area of research could involve introducing a more diverse model for Web pages as opposed to just large and small. A Web page could be considered to be composed of a number of cacheable (and uncacheable) objects each of varying sizes. A more complex model for Webpages would lead to more options for an evolutionary algorithm. Finally, if considering the potential real-world application of an evolutionary algorithm for controlling partition sizes, it would be important to conduct a detailed examination of the overhead associated with utilizing the algorithm (a factor that was ignored in this research).

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Athena, V. (2002) Evolutionary Techniques for Web Caching. *Distributed and Parallel Databases*, **11**, 93-116. https://doi.org/10.1023/A:1013385708178

[2] Calzarossa, M.C., Massari, L. and Tessera, D. (2016) Workload Characterization: A Survey Revisited. *ACM Computing Survey*, **48**, Article No. 48.

https://doi.org/10.1145/2856127

[3] Arlitt, M., Cherkasova, L., Dilley, J., Friedrich, R. and Jin, T. (1999) Evaluating Content Management Techniques for Web Proxy Caches. *ACM SIGMETRICS Performance Evaluation Review*, **27**, 3-11. https://doi.org/10.1145/346000.346003

[4] Berger, D.S., Beckmann, N. and Harchol-Balter, M. (2018) Practical Bounds on Optimal Caching with Variable Object Sizes. *ACM SIGMETRICS Performance Evaluation Review*, **46**, 24-26. https://doi.org/10.1145/3292040.3219627

[5] Veliskakis, M., *et al.* (2005) Domproxy: Enabling Dynamic-Content Front-End Web Caching. *Proceedings of the* 10*th International Workshop on Web Content Caching and Distribution*, Sophia Antipolis, 12-13 September 2005, 56-61.

[6] Kvaternik, K., Llorca, J., Kilper, D. and Pavel, L. (2016) A Methodology for the Design of Self-Optimizing, Decentralized Content-Caching Strategies. *IEEE*/*ACM Transactions on Networking*, **24**, 2634-2647. https://doi.org/10.1109/TNET.2015.2478059

[7] Nguyen, H.V., Iacono, L.L. and Federrath, H. (2019) Mind the Cache: Large-Scale Explorative Study of Web Caching. *Proceedings of the* 34*th ACM*/*SIGAPP Symposium on Applied Computing*, New York, NY, April 2019, 2497-2506. https://doi.org/10.1145/3297280.3297526

[8] Plumley, B. and Hurley, R.T. (2016) Effectiveness of Load Balancing in a Distributed Web Caching System. *Proceedings of the* 7*th International Conference on Computer Modelling* (*ICCM*2016), Berkeley, CA, 1-4 August 2016, 46-60.

[9] Hurley, R.T. and Li, B.Y. (2008) A Performance Investigation of Web Caching Architectures. *Proceedings of the* 2008 *C^3S^2E Conference*, 12-13 May 2008, 205-213. https://doi.org/10.1145/1370256.1370291

[10] Arlitt M., Friedrich, R. and Jin, T. (1999) Performance Evaluation of Web Proxy Cache Replacement Policies. *Lecture Notes in Computer Science*, **1469**, 193-206. https://doi.org/10.1007/3-540-68061-6_16

[11] Hurley, R.T., Feng, W. and. Li, B.Y. (2003) Performance Benefits of Partitioning in a Web-Caching Environment. *Proceedings of* 16*th International Conference on Computer Applications in Industry and Engineering*, Las Vegas, Nevada, November 11-13, 2003, 64-69.

[12] Bonino, D., Corno, F. and Squillero, G. (2003) A Real-Time Evolutionary Algorithm for Web Prediction. *Proceedings of the IEEE*/*WIC International Conference on Web Intelligence*, Halifax, NS, 13-17 October 2003, 139-145. https://doi.org/10.1109/WI.2003.1241185

[13] Young, G. (2012) On the Use of Evolutionary Programming to Dynamically Alter Cache Sizes. Master's Thesis, Trent University, Peterborough, ON, Canada.

[14] Plumley, B. (2015) An Investigation of Load Balancing in a Distributed Web Caching Systems. Master's Thesis, Trent University, Peterborough, ON, Canada.

[15] Bodnarchuk, R.R. and Bunt, R.B. (1991) A Synthetic Workload Model for a Distributed System File Server. *ACM SIGMETRICS Performance Evaluation Review*, San Diego, CA, April 1991, 50-59. https://doi.org/10.1145/107972.107978

[16] Butkiewicz, M., Madhyastha, H.V. and Sekar, V. (2011) Understanding Website Complexity: Measurements, Metrics, and Implications. *Proceedings of the* 2011 *ACM SIGCOMM Conference on Internet Measurement Conference*, Berlin, Germany, November 2011, 313-328. https://doi.org/10.1145/2068816.2068846

[17] Hasslinger, G., Ntougias, K., Hasslinger, F. and Hohlfeld, O. (2016) Performance Evaluation for New Webcaching Strategies Combining LRU with Score Based Ob-

ject Selection. 2016 28*th International Teletraffic Congress* (*ITC* 28), Würzburg, Germany, 12-16 September 2016, 322-330. https://doi.org/10.1109/ITC-28.2016.150

[18] Li, B.Y. (2002) An Investigation of Partitioned Caching in the World Wide Web. Master's Thesis, Trent University, Peterborough, ON, Canada.

[19] Rabl, T. and Jacobsen, H.-A. (2017) Query Centric Partitioning and Allocation for Partially Replicated Database Systems. *Proceedings of the* 2017 *ACM International Conference on Management of Data*, New York, NY, May 2017, 315-330. https://doi.org/10.1145/3035918.3064052