computers

MDPI

Article

On Implementing Autonomic Systems with a Serverless Computing Approach: The Case of Self-Partitioning Cloud Caches †

Edwin F. Boza [1], Xavier Andrade [1], Jorge Cedeno [1], Jorge Murillo [2], Harold Aragon [1], Cristina L. Abad [1,*] and Andres G. Abad [1]

[1] Escuela Superior Politecnica del Litoral, Guayaquil EC090112, Ecuador; eboza@espol.edu.ec (E.F.B.); xaandrad@espol.edu.ec (X.A.); jlcedeno@espol.edu.ec (J.C.); haragon@espol.edu.ec (H.A.); agabad@espol.edu.ec (A.G.A.)

[2] University of Massachusetts Amherst, Amherst, MA 01003 USA; jrmurillo@cs.umass.edu

* Correspondence: cabadr@espol.edu.ec

† This paper is an extended version of our paper published in the Workshop on Self-Aware Computing (SeAC), held in conjunction with Foundations and Applications of Self* Systems (FAS* 2019)

Received: 12 January 2020; Accepted: 23 February 2020; Published: 26 February 2020

Abstract: The research community has made significant advances towards realizing self-tuning cloud caches; notwithstanding, existing products still require manual expert tuning to maximize performance. Cloud (software) caches are built to swiftly serve requests; thus, avoiding costly functionality additions not directly related to the request-serving control path is critical. We show that serverless computing cloud services can be leveraged to solve the complex optimization problems that arise during self-tuning loops and can be used to optimize cloud caches for free. To illustrate that our approach is feasible and useful, we implement SPREDS (Self-Partitioning REDiS), a modified version of Redis that optimizes memory management in the multi-instance Redis scenario. A cost analysis shows that the serverless computing approach can lead to significant cost savings: The cost of running the controller as a serverless microservice is 0.85% of the cost of the always-on alternative. Through this case study, we make a strong case for implementing the controller of autonomic systems using a serverless computing approach.

Keywords: self-tuning; cloud computing; serverless computing; autonomic controller

1. Introduction

Application-controlled cloud caches implemented with fast in-memory key-value stores, like Redis and Memcached, have become ubiquitous in modern web architectures [2]. Content providers use caches to reduce latency and increase throughput, increase user engagement and profits, and reduce infrastructure costs [2,3]. Proper configuration and tuning of these caches is critical, as sub-optimal configurations lead to increased miss rates and a resulting penalty in end-to-end performance, negatively affecting the business goals: It has been reported by Amazon that a 100 ms latency penalty can lead to a 1% sales loss, and by Google that an additional 400 ms delay in search responses can reduce search volume by 0.74% [3].

A time-tested way to improve cache performance is to optimally partition the cache [1,2,4–15], for example, by dynamically partitioning the total memory between users or applications. However, current cloud caches support only static partitioning, while others provide no control over the partitioning. Redis (Remote Dictionary Server) is an example of the former, while Memcached is of the latter. While several solutions targeting cloud caches have been proposed [9–14], these have not been added to industry-grade software caches due to performance concerns.

In this paper, we present a serverless computing architecture using the function-as-a-service model to optimize cloud caches for free. We implement this solution in SPREDS, a self-partitioning REDiS. SPREDS leverages modern data structures and statistical sampling methods to efficiently obtain online estimates of the real miss rate curves (MRCs) [16]. The backend performance profiles and MRCs are combined into a utility function to maximize.

The resulting optimization problem is solved outside the cache using a serverless computing approach. Our experimental results show that the performance overhead due to monitoring the cache is low and that implementing the autonomic controller as a serverless microservice is feasible and useful. We present cloud cost calculations that show that the invocations to the controller are either free or very cheap, with the cost of running the controller as a serverless microservice being just 0.85% of the cost of the always-on alternative. Additionally, the optimization problem is solved outside of the machine running the cache and does not consume the resources of the caching nodes.

Using SPREDS as a case study, we argue that the proposed approach should be considered in future autonomic and self-* systems, as it is a low-cost and low-overhead way to calculate complex adaptation decisions applicable to systems running on public cloud providers. We end by outlining some challenges in implementing this vision and discuss how to address them.

1.1. Contributions and paper roadmap

This work makes the following contributions:

1. We survey the most current research in the domain of self-tuning cloud caches and study the designs used by prior approaches as a way to motivate the need for a more modern, cheaper, cloud-native solution (Section 2).
2. We re-visit the memory partitioning problem and model it as a mathematical optimization problem with restrictions that are specific to the multi-instance cache on a shared node studied in this paper (Section 3).
3. We present a novel design for implementing autonomic cloud caches that leverages serverless computing cloud offerings to implement the autonomic controller (optimization module) at a low cost (Section 4).
4. We show the feasibility of our approach through implementing SPREDS, a real implementation of our design using Redis and AWS Lambda (Section 4).
5. We present real experimental results and a corresponding cost analysis that validate the usefulness of our proposal (Section 5).
6. We make a case for adopting our serverless approach in other autonomic, self-tuning systems and study the challenges in realizing this vision (Section 6).

1.2. Relation to Prior Work by the Authors

The work herein builds upon prior contributions from our group [1,2,16], as described herein.

In a work-in-progress paper [2] presented at the 2017 ACM/SPEC International Conference on Performance Engineering (ICPE), we formalized the memory partitioning problem for the case of multi-instance cloud caches with heterogeneous backends. SPREDS is based on an extension of that model and one of the solvers in SPREDS is based on the algorithm introduced in [2].

In a workshop paper [16] presented at the Adaptive and Reflective Middleware (ARM) workshop, which was held in conjunction with the 2017 ACM/IFIP/Usenix Middleware Conference, we described how to instrument Memcached to obtain miss rate curves (MRCs) at a low performance cost. For the current work, we adapted that design and ported it to Redis. While the engineering details differ, there are no further scientific contributions in the monitoring module, and thus, we have not listed the lightweight monitoring as a contribution in the current paper.

Finally, the present paper is an extension of our paper from the Workshop on Self-Aware Computing (SeAC), held in conjunction with Foundations and Applications of Self* Systems (FAS* 2019) [1]. As such, the contributions listed in this paper are the same ones as in [1]. However, we have extended our SeAC work with additional use cases, more detailed information on our mathematical optimization model and the genetic algorithm we proposed to solve it, a considerably larger literature review, and more experimental results.

1.3. Threats to Validity

We assume that the mathematical optimization problem to find the optimal memory partitioning can be solved in a reasonable amount of time (e.g., takes no more than 25% of the time between adaptation cycles). If solving the optimization problem takes too long, the solution proposed in this paper is not applicable.

Our solution relies on estimated miss rate curves, as getting real MRCs with a low performance overhead is—to the best of the state-of-the-art knowledge in caching [17]—not possible. However, the results we present from real cloud experiments show that the accuracy of the MRCs is good enough for our purposes: SPREDS can improve performance over a static partitioning approach. Nevertheless, it is possible that there may exist workloads for which the estimated curves are not good proxies of the real ones.

Automatic parameter tuning depends on being able to monitor dynamic behavior at a granularity that is useful for making informed predictions of the impact of changing specific parameters. Our solution leverages recent advances in efficient MRC estimation [16] that apply only to the caching domain. An alternative approach to be considered for future work is constructing a utility function that relies only on the metrics and information that can be obtained from system logs; this would be a zero-overhead approach, as much useful information from each system component is typically already being logged, stored, and monitored—following a modern DevOps mindset [18,19].

Finally, we argue for using a serverless computing approach in our design and present cost estimates that are considerably cheaper than the alternative of an always-on service. Whether this is actually true or not in practice depends on (1) how frequently the optimization service is invoked and (2) the costs of each of the cloud services used in its implementation. We believe the serverless approach may be the cheapest option for small and medium-sized organizations in the near future, but cheaper alternatives may arise in the long run or may already be available for larger organizations, for which an always-on service is likely a better alternative.

2. Background, Motivation, and Related Work

Key-value stores are primitive databases that support efficient insertion and lookup of data indexed by user-defined keys. In-memory key-value stores work like remote hash tables or dictionaries and can answer requests with very high throughput and low latency. At the time of this writing, the most common in-memory key-value store software products are Redis (https://redis.io) and Memcached (https://memcached.org) [20]. These products are frequently used as caches at the backend of modern web and mobile applications. Caches implemented with in-memory key-value stores are not transparent caches; the cache is invoked explicitly in the applications, serving complex business logic workflows. In this paper, we study Redis because it is the most popular key-value store as of December 2019 [20].

We show a common use case of cloud caches in Figure 1: a system that stores information—like product inventory, user profiles, and session information—in different storage backends. There is a frontend that aggregates information from these backends and presents it to the user. To improve latency and reduce pressure on the storage backends when serving user requests, the frontend typically first contacts a caching layer seeking the required information. If the data being sought is not stored in the cache—a cache miss—the application needs to contact the specific storage backend, get the data, and return it to the user. The application then contacts the cache to store a copy of the data so that

future requests can be served directly from the cache. Each of these storage backends have their own unique performance profiles. For example, the databases supporting complex queries are much slower than object storage devices that store image files. The following is a small list that exemplifies the type of things being cached in the backends of web or mobile applications:

- User profiles, where profiles contain data pulled from several other backend systems;
- Tracking information, like user engagement counters;
- User avatars or other images shown in the frontend to the end users;
- Business reports resulting from querying multiple tables in an Online Transaction Processing (OLTP) database;
- Session information, like the per-user application system state;
- User status updates in social networks; users read the updates of their "friends".

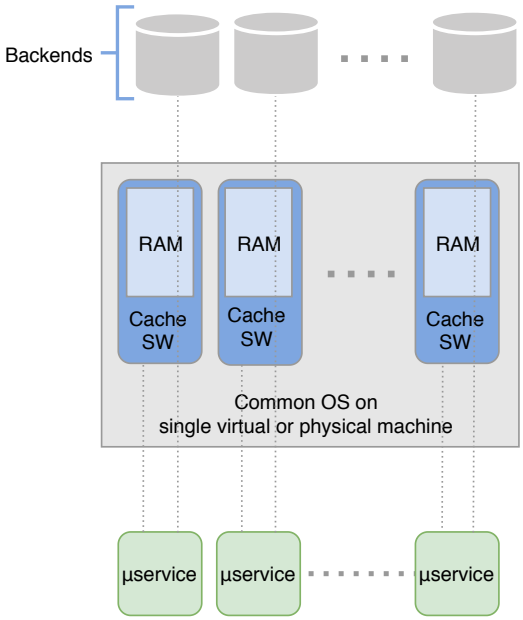

Figure 1. Illustration of the motivating architecture used in this paper. The figure shows a web or mobile application with a microservices architecture. There is a caching software (SW) layer that speeds up accesses to data stored in several heterogeneous backends.

One important factor in the performance of a cache is how much memory it has available to store objects: The larger the cache, the higher the percentage of objects from the storage backend that it can hold and thus, the higher the likelihood of a request being a hit and not a miss.

The cache's eviction policy determines which objects are removed from the cache to make space for new objects being added. Cloud caches support several eviction algorithms, with least recently used (LRU) or some variant of this algorithm being the most commonly used one as it is known to perform well for a wide variety of workloads. LRU assumes that recent past behavior is a good predictor of future behavior. When an object is to be evicted, it selects the object that has been used least recently because this is the one that it predicts is the least likely to be re-used in the near future. By default, Redis uses a randomized version of LRU [21] that samples x objects and evicts the one that was accessed least recently; x is configurable and set to five by default. Since version 3, the LRU implementation of Redis also takes a pool of good candidates for eviction.

The other important factor influencing the performance of a cache is the workload characteristics [22,23] like the skewness of the distribution of the popularity of the objects and the degree of temporal locality present in the accesses to the objects in the cache. These workload characteristics are application-dependent and may be dynamic; thus, the caches must be able to self-adapt to these changes.

2.1. How Important Is (Optimal) Memory Partitioning in Cloud Caches?

To make the best use of CPU resources, modern microservices architectures favor shared-nothing and stateless approaches to distributed systems. A common scenario is for multiple Redis instances to be co-located in the same machine (see Figure 1). Each of these instances serves a different workload or application and can be configured independently. In this scenario, the machine's memory becomes an important resource that must be adequately partitioned between the cache instances.

We surveyed recent papers tackling the problem of self-partitioning cloud caches [9–14] and found that intelligent cloud cache partitioning can lead to significant performance gains: Depending on the service-level objectives and resulting utility function, these improvements can be in terms of higher throughput, higher cache hit rate, and reduced end-to-end latency. Tables 1 and 2 summarize our observations. Based on the results presented by the prior studies and on our own observations, we posit that smart and adaptive memory partitioning in caches supporting different workloads leads to significant performance gains. Sadly, memory partitioning in Redis is currently only static and manual. The case of Memcached is even worse as it gives more memory to the application issuing more requests, regardless of whether this is good for the overall system [24].

Table 1. Summary of the recent work in the domain of self-partitioning cloud caches. For each of the papers, we highlight one result demonstrating the achievable performance gains. Prior solutions differ due to differences in methods, service-level objectives, workloads, and experimental configurations.

System	Open Source?	Goal	Result
Moirai [9]	Prototype	Supports diverse SLOs	Overall throughput increases by more than 2.5× in a multi-tenant data center.
Dynacache [10]	No	Minimize miss ratio	Reduces number of misses by more than 65%.
LAMA [11]	No	Minimize miss ratio	Reduces average miss ratio by 41.9%, saving 40.8% of memory space.
Cliffhanger [12]	No	Minimize miss ratio	Reduces number of cache misses by 36.7%.
RobinHood [13]	Testbed	Minimize request P99	Meets 150 ms P99 goal 99.7% of time (vs. 70% of time for next best policy).
Memshare [14]	Simulator	Minimize miss ratio	Increases combined hit rate from 84.7 to 90.8%.

Table 2. Architecture of the self-partitioning caches proposed in recent literature. MR: miss rate; MRC: miss rate curve; LP: Linear programming; SPREDS: Self-Partitioning REdiS.

System	Location of Metrics Engine (Monitoring Agent, MA), and Location of Optimizer (Autonomic Controller, AC)
Moirai	MA: Local hypervisor module analyzes workloads on system. AC: Centralized controller, external to cache; shared at data-center level.
Dynacache	MA: Offline external; heavyweight stack distance algorithm. AC: In-cache LP solver that assumes convexity of MRCs.
LAMA	MA: In-cache metrics (independent thread). AC: In-cache algorithm (dynamic programming algorithm).
Cliffhanger	MA: In cache; approximates MR gradients w/ shadow queues. AC: In-cache; incremental *cliffhanger* algorithm.
Memshare	MA: In cache *arbiter* approximates MR gradients. AC: In-cache arbiter uses MR gradients to incrementally adjust partition sizes.
RobinHood	MA: External server; shared between applications. AC: Distributed controller; one per caching server.
SPREDS	MA: Local to cache; external process. AC: Pay-per-use service on a serverless computing platform.

2.2. Architectures for Self-Partitioning Cloud Caches

Self-adjusting capabilities that involve solving complex optimization problems can impose unacceptable performance overheads on the system being tuned. We argue that a serverless architecture approach can be used to overcome this limitation. To better explain why this is the case, we first study the architectures used in prior self-partitioning caches. We summarize their architectural decisions regarding where to locate the monitoring engine and the autonomic controller in Table 2 and analyze the limitations of these architectures next.

Prior projects have used one of the following approaches to limit the performance overhead introduced by solving the optimization problem:

A1: Make small, gradual changes to the memory allocations exploring the configuration space to find the optimal partitioning. This approach removes the CPU cost of solving the optimization problem but introduces frequent resizing costs as the internal state and data structures of the cache need to be modified for each small step in the exploration of the configuration space. Cliffhanger [12] is an example of a system that makes small, continuous, costly changes.

A2: Simplify the problem by limiting the number of partitions and making other (unrealistic) assumptions about the workload, thus ensuring that solving the optimization problem becomes tractable without incurring high computation costs. This approach has been suggested as a way to solve the optimization problem within the cloud cache but without consuming excessive CPU resources, which would slow down the cache requests. Dynacache [10] is an example that simplifies the problem to make it more tractable.

A3: Take the solving of the optimization problem outside of the caching software and move it to an external controller. This is the most common approach in the literature of self-adaptive systems (e.g., see [2,25,26]).

Approach A3 is the better option because it does not incur frequent resizing costs (limitation of A1), nor does it make unrealistic assumptions about the workload (limitation of A2). For this reason, we opt for approach A3 in SPREDS.

It should be noted that while A3 overcomes the performance and accuracy limitations of A1 and A2, it may incur additional costs related to running the external controller. We are not aware of prior work comparing the monetary costs of running an external controller. These costs can differ depending on the architectural approach used to implement A3 in a real system:

(a) Always-on shared service: An always-on shared cloud adaptation service can be offered for free or at a reasonable cost by the cloud or a third-party provider. The costs are shared by tenants or absorbed by the provider seeking a competitive advantage. An example of a recent product in this domain is the self-indexing service for the Microsoft Azure SQL DB [27]. However, this approach lacks flexibility and does not encourage innovation in the tuning algorithms (the service provider controls the algorithm and tenants cannot test improved adaptation algorithms). Furthermore, some utility functions—like those that seek to minimize operational costs—may go against the provider's business interests and the providers would not offer them to their tenants.

(b) Always-on client-managed service: The cloud tenant runs the controller as one more always-on service in their system. For example, in the database domain, OtterTune [25] has been proposed as an external database tuning system that can tune the performance of databases as well as external human experts. The costs of the always-on client-managed service approach can be too high for small or medium organizations if the service is infrequently used. In addition, it has the added overhead of having to manage an additional online service and thus, constitutes an option only suitable for large enterprises.

(c) Serverless microservice: The approach argued for in this paper is deploying the controller as a serverless microservice using a Function-as-a-Service (FaaS) offering like Amazon Web Services (AWS) Lambda (https://aws.amazon.com/lambda/) or Azure Functions (https://azure.microsoft.com/en-us/services/functions/). FaaS offerings are gaining increasing attention in the community as they let tenants run code without provisioning or managing servers and pay only for the compute time they consume [28]. The solver is an external process owned by the client and implemented using a serverless architecture. This on-demand approach avoids service over-provisioning and reduces the costs of operating the controller. Furthermore, it lets the tenants tailor the utility function and optimization-solving algorithm according to their specific needs.

The details of our proposal are presented in Section 4. In Section 5, we experimentally validate the approach and present a cost analysis that shows that the always-on client-managed service approach is much more expensive than the serverless approach advocated for in this paper.

2.3. Why Aren't Current Cloud Caches Already Self-Partitioning?

Some of the projects described in Table 1 involved collaborations with industry—namely, Facebook, Akamai, and Microsoft. It is possible that these (and other) companies have self-partitioning cloud caches being used in production. However, these adaptation mechanisms have not been added to the open source versions of Redis or Memcached.

Approaches that gradually explore the configuration space hoping to find an optimal solution (e.g., [12]) incur a high penalty when the cost of reconfiguration is not negligible, as has been observed in the current implementation of the resizing mechanism of Redis [2]. Furthermore, these solutions are tailored for a specific solution and do not scale to self-tuning other knobs of the caching software. A better method involves utility-driven approaches, which are flexible and can support diverse service-level objectives. However, these are deemed unscalable due to the high CPU consumption of the algorithms used to solve the optimization problem [29]. We believe the serverless architecture proposed in this paper can overcome both challenges and can facilitate adding self-adaptation functionality to cloud caches and other software in the near future.

2.4. Other Related Work

Earlier in this Section, we analyzed the most relevant prior work in self-partitioning cloud caches; the results of our analysis were presented in Tables 1 and 2. In this subsection, we discuss other prior research that is related to SPREDS.

Other uses of workload-driven partitioning schemes for caches include partitioning flash-based caches into hot/cold areas to support efficient data compression [30] or to determine the right number of replicas of each partition [31]. In addition to improving performance, others have included the notions of fairness, isolation, and strategy proofness in their partitioning schemes [24,32]; SPREDS was designed for the case in which the applications belong to the same tenant and do not try to game the system. Another consideration that can be incorporated into the optimization problem is the penalty associated with memory reallocation during partitioning cycles; solutions like the one used in pRedis [33], that factor this into the optimization problem, are orthogonal to the architectural approach proposed in this paper.

A few self-tuning database products and services have been proposed by academia and industry. OtterTune [25] is a tuning service for MySQL and PostgreSQL that automates the process of finding good settings for a database's configuration, reusing training data gathered from previous tuning sessions. Das et al. [27] describe an auto-indexing service for Microsoft Azure SQL Database. Oracle recently released their Autonomous (cloud) Database (https://www.oracle.com/database/autonomous-database.html) and ScyllaDB offers a database with limited self-managing properties (https://www.scylladb.com/); however, the latter is a rule-based tuning solution [34].

Idreos et al. [35] recently introduced the concept of design continuums for the data layout of key-value stores and present a vision of self-designing key-value stores that automatically choose

the right data layout for a specific workload and memory budget. This is a related but different problem than the one used as a case study in this paper; both self-* approaches can co-exist in the same key-value store.

In general, the idea of offering adaptations "as a service", as a path to making autonomic systems a reality, has been argued before [2,26]. However, a traditional always-on service makes sense when the service is provided by the cloud or a third party provider, or for large organizations. In this paper, we propose a variant of this idea, with a serverless microservice architecture, and provide a proof-of-concept implementation highlighting its usefulness.

3. The Memory Partitioning Problem

We consider a system where a virtual or physical machine hosts n instances of the caching software, each serving a different application as depicted in Figure 1; these applications compete for the allocation of the total memory, M. The amount of memory assigned to each application i is denoted by m_i. Our model also works for a multitenant architecture, as long as the different applications sharing the cache belong to the same organization. Table 3 contains a reference of the parameters used in the model definition.

Table 3. Parameters used in the model definition.

Parameter	Description
n	Number of instances of the cache running on the system.
i	Identifies a specific application or workload.
M	Total system memory.
m_i	Memory assigned to application i.
$\underline{m_i}$	Minimum memory assignment for application i.
bd_i	Access latency of the backend system, including the time to process a cache miss.
cd_i	Access latency of the caching system.
U_i	Individual utility function of application i.
w_i	User-defined weight for application i.
f_i	Access frequency of application i.
EAT_i	Effective access time of application i.

When application i needs some data, it first looks for it in the cache. If the sought data is not there—i.e., a cache miss occurs—the application obtains the desired information from its corresponding storage backend. Each backend has its own performance profile; i.e., its own average latency to access the backend bd_i. For example, a database server used to build user profiles may likely be slower answering requests than a service that generates unique user identity avatars based on the client's username or IP address (e.g., like Github's identicon).

After a cache miss, the data is typically inserted in the cache (after making space for it by evicting less valuable data, if necessary) so that it will be available at the cache for future requests. However, as these are explicit—not transparent—caches, the application is free to implement a more complex admission logic.

We consider memory as the only shared resource, ignoring the sharing of CPU. In-memory key-value stores are memory- and not CPU-bound. This has been reported by Redis (http://redis.io/topics/faq) and observed in real Memcached deployments [12].

In this work, the goal is Pareto efficiency: fully utilize the memory, compute the ideal memory allocation $\mathbf{m} = [m_1, \ldots, m_n]$, and achieve the highest overall utility given individual utility functions, U_is, and the total memory constraint M. This can be expressed as the following optimization problem:

$$\begin{aligned} \underset{\mathbf{m}}{\text{maximize}} \quad & \mathcal{F}(\mathbf{m}) = \sum_{i=1}^{n} w_i U_i(m_i), \\ \text{subject to} \quad & \sum_{i=1}^{n} m_i \leq M, \\ & m_i \geq \underline{m_i}, i = 1, ..., n, \end{aligned} \tag{1}$$

where $U_i(m_i)$ is the utility function of application i as a function of its assigned memory m_i and configured w_i (which lets us indicate that one application is more or less important). $\underline{m_i}$ is the minimum memory assignment for application i; it can be set to zero if it is acceptable for the system to decide not to cache the objects of some application.

We assume a non-adversarial model in which the applications are not trying to game the system. This is a reasonable assumption when all the applications belong to the same cloud client. Given that we consider a non-adversarial model, we do not seek strategy proofness [24]. Some application could be able to issue workloads that lead to a higher memory assignment to said application, but this would be at the cost of reduced overall system performance.

For the utility function, we consider improving average access latency to objects in the cache to be our most important optimization metric. For eviction algorithms that fulfill the inclusion principle [36]—e.g., LRU—giving more memory to the cache means that the hit rate will either stay the same or will improve. Thus, the way to optimize the performance for a single application is to give it as much memory as possible. As the memory is a shared resource that the applications are competing for, we devise a utility function that combines each of the utilities as a function of how much memory we have assigned to the application $U_i(m_i)$, weighed by a user-defined weight (w_i) so that the tenant can declare that improving the performance of one application is more important than for others.

We posit that improving the hit rate of one application may not be as useful as improving the hit rate of another application due to differences in the performance profiles of the corresponding backends. For example, all other things being equal, if one application has a slow backend (e.g., one that processes slow, multi-table, Standard Query Language (SQL) queries), increasing the hit rate of the cache serving that application is more useful than increasing the hit rate of a cache serving a fast backend (e.g., a NoSQL database that stores session information). For this reason, for each application we calculate its effective access time (EAT) and weigh it by the inverse of the application's access frequency (f_i). The formula for the EAT is calculated using the classic approach devised for two-level memory systems [37], where we consider the access latency to the two levels—the cache and the backend storage—and the probability that an object or data item will be found in the fast memory tier, which in our case is given by the cache hit rate. The access latency to these two levels is denoted by cd_i (object latency in the caching system) and bd_i (object latency in the backend system, including the time to process a cache miss). For a given application, the cache hit rate is a function of how much memory the application has been assigned $h_i(m_i)$ and can be directly obtained from the application's miss rate curve (MRC). In other words, the EAT_i is the time that it takes, on average, to access an object in application i. We define the following per-application utility function:

$$\begin{aligned} & U_i(m_i) = -f_i \times EAT_i(m_i), \text{ where} \\ & EAT_i(m_i) = h_i(m_i) \times cd_i + [1 - h_i(m_i)] \times bd_i, \end{aligned} \tag{2}$$

where $h_i(m_i)$ is the hit rate of application i as a function of assignment m_i, and f_i is the frequency of requests of i.

3.1. Solving the Optimization Problem

Consider problem (1): If the U_is are quasi-linear, the resulting optimization can be solved by solving a sequence of feasibility problems, with a guaranteed precision of ϵ in $\lceil \log_2 R/\epsilon \rceil$ iterations, where R is the length of the search interval. If the U_is are concave, we have a convex optimization

problem easily solved with off-the-shelf solvers; for example, using gradient-based methods. When the U_is are discontinuous, non-differentiable, or non-convex, alternative approaches are required.

One alternative is to use a probabilistic search, in which a model generates candidate points in the search for an optimum. An adaptive mechanism may be added to the generative model to improve the performance of the sequentially generated candidates. We proposed one such approach to memory partitioning in earlier work [2] and implemented it in SPREDS (see Algorithm 1). This genetic algorithm works for any partitioning problem as it makes no assumption of the shape of the utility curves. We next describe this approach, which is one of the two solvers implemented in SPREDS.

Let $\mathbf{x} = [x_1, \ldots, x_n]$ with $x_i = (m_i - \underline{m}_i)/(M - \sum_i \underline{m}_i)$, satisfying $\sum_i x_i = 1$ and $0 < x_i < 1$. We assume that the variability of $\mathbf{x}$ can be well modeled by a Dirichlet distribution. The Dirichlet distribution $Dir(\mathbf{x}|\alpha)$ has a density function:

$$f(\mathbf{x}|\alpha) = \frac{\Gamma(\sum_{i=1}^{n} \alpha_i)}{\prod_{i=1}^{n} \Gamma(\alpha_i)} \prod_{i=1}^{n} x_i^{\alpha_i - 1}, \tag{3}$$

expected value $\mathbb{E}[x_i] = \alpha_i / A$, and variance $\mathbb{V}[x_i] = \alpha_i(A - \alpha_i)/(A^2(A+1))$, where $A = \sum_i \alpha_i$. We model the variability with a Dirichlet distribution $Dir(\mathbf{x}|\alpha)$ with parameter vector $\alpha = [\alpha_1, \ldots, \alpha_n]$. Then, we define

$$\tilde{\mathcal{F}}(\mathbf{x}) = \mathcal{F}\left((M - \sum_i \underline{m}_i)\mathbf{x} + \underline{\mathbf{m}}_i\right), \tag{4}$$

where $\underline{\mathbf{m}}_i = [\underline{m}_1, \ldots, \underline{m}_n]$.

We propose the following general approach, inspired by evolutionary strategies, for solving problem (1) in the case of non-convex and non-quasi-linear function U_is. We begin by setting α_i to $1/n$ for all i. We then generate K points $\mathbf{x}_k^\star|\alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$. Note that points generated in this way satisfy all restrictions of problem (1).

Using points in set $\{\mathbf{x}_k^\star\}_{k=1}^K$, we construct the following prior mixture density for α:

$$g(\alpha; \{\mathbf{x}_k^\star\}) = \frac{1}{Z} \sum_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \phi_{\mathbf{x}}(\alpha), \tag{5}$$

where Z is a normalization constant and $\phi_{\mathbf{x}}$ is a non-negative function with finite mass concentrated around $\mathbf{x}$ (e.g., a radial basis function centered at $\mathbf{x}$). We proceed by sampling α from g and generating points $\mathbf{x}|\alpha_\gamma \sim Dir(\mathbf{x}|\alpha_\gamma)$, where α_γ is the vector with elements $\gamma\alpha_i$ for $\gamma > 1$. Note that while random variables $\mathbf{x}|\alpha$ and $\mathbf{x}|\alpha_\gamma$ have the same expected value, γ has the effect of reducing the variance of $\mathbf{x}|\alpha_\gamma$ by a factor of γ^{-1}.

The above generative procedure corresponds to the following Bayesian hierarchical structure:

$$\begin{aligned} \alpha &\sim G(\alpha; \{\mathbf{x}_k^\star\}) \text{ and} \\ \mathbf{x}|\alpha_\gamma &\sim Dir(\mathbf{x}|\alpha_\gamma), \end{aligned} \tag{6}$$

where G is the distribution function corresponding to mixture density g.

Our method proceeds by alternatively generating parameters α—the exploration stage—and generating J points of $\mathbf{x}$ conditioned on α_γ—the exploitation stage. The prior distribution for α is then updated using the K best cumulatively observed points, $\mathbf{x}_k^\star$s, and the procedure is repeated until a satisfactory solution is found. Algorithm 1 provides the details of the proposed procedure.

We also implemented a hill-climbing algorithm combined with a LookAhead approach [7]. We added the LookAhead approach so that this algorithm can deal with non-convex utility curves; the genetic algorithm is applicable to any shape of the utility function. SPREDS considers the number of partitions and chooses the solving method depending on the complexity of the problem. Based on experimental results presented in Section 4.3, we heuristically choose between both algorithms as

follows. For deployments with a few partitions ($i \leq 7$), we use the proposed genetic algorithm. For deployments with a larger number of partitions, we use the hill-climbing solver. Intelligently choosing the solver is outside of the scope of this paper [2,38].

Algorithm 1: Probabilistic adaptive search

Input: Functions U_is; number K of points to use; number J of rounds; function $\phi_{\mathbf{x}}$
Output: Best point $\mathbf{x}^*$, such that $\tilde{\mathcal{F}}(\mathbf{x}^*) \geq \mathbf{x}$ for every point $\mathbf{x}$ generated

$\alpha_i := 1/n$ for $i = 1 \ldots, n$
Generate $\mathbf{x}_k^\star | \alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$
repeat
  Generate $\alpha \sim G(\alpha; \{\mathbf{x}_k^\star\})$
  for $j = 1, \ldots, J$ **do**
    Generate $\mathbf{x}|\alpha_{\gamma^{(j)}} \sim Dir(\mathbf{x}|\alpha_{\gamma^{(j)}})$
    if $\tilde{\mathcal{F}}(\mathbf{x}) > \min_k \{\tilde{\mathcal{F}}(\mathbf{x}_k^\star)\}$ **then**
      $\{\mathbf{x}_k^\star\} := \mathbf{x} \cup \{\mathbf{x}_k^\star\} \setminus \arg\min_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$
until *Satisfactory solution* $\mathbf{x}^\star = \arg\max_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$ *is found*;
return $\mathbf{x}^\star$

4. Design and Implementation of SPREDS

We map our design of a self-partitioning cloud cache to a MAPE-K loop [39] with its corresponding monitor, analyze, plan, and execute functions.

The **monitor**ing component runs in the same machine as the cache and can be part of the cache or an external sidecar [40] microservice. This component uses statistical sampling techniques to minimize the monitoring impact. The monitor stores every N observations on a configurable cloud location. When a new set of observations is stored, a *calculate new adaptation* event is triggered. The size of the set of observations, N, is configurable and lets the user decide how frequently the system adaptations should be triggered—i.e., how frequently to calculate the optimal memory configuration and to re-partition the cache memory if necessary. Criteria for choosing N include the cache throughput and how dynamic the application workloads are.

The controller is implemented as a serverless microservice. It performs the **analyze** phase when launched in response to a *calculate new adaptation* event. In the SPREDS implementation, this microservice solves the mathematical optimization to determine how to best partition the cache memory according to the specific set of observations captured by the monitor.

The controller generates an adaptation **plan** to implement the solution and stores it on a specific cloud storage location. The action of storing the adaptation plan on the cloud storage triggers an *execute adaptation* event, which is then received by the cache. When **execute**d, this plan re-partitions the cache according to the most current (optimal) solution.

In our design, the **knowledge source** is one or more locations (e.g., buckets or directories) in a cloud storage system managed by the provider. This is where the system stores the captured metrics, monitoring data, and adaptation plan. Alternatively, one or more specialized databases could be used for this purpose. For example, the Prometheus (https://prometheus.io/) time series database is commonly used in cloud-native systems to store metrics analyzed by DevOps teams.

To validate our design, we implemented SPREDS as a modified version of the Redis key-value store that leverages several AWS services to periodically self-partition the memory seeking to maximize overall system utility, as depicted in Figure 2. The current implementation of SPREDS extends Redis version 4. The Amazon Web Services products used in SPREDS are described in Table 4.

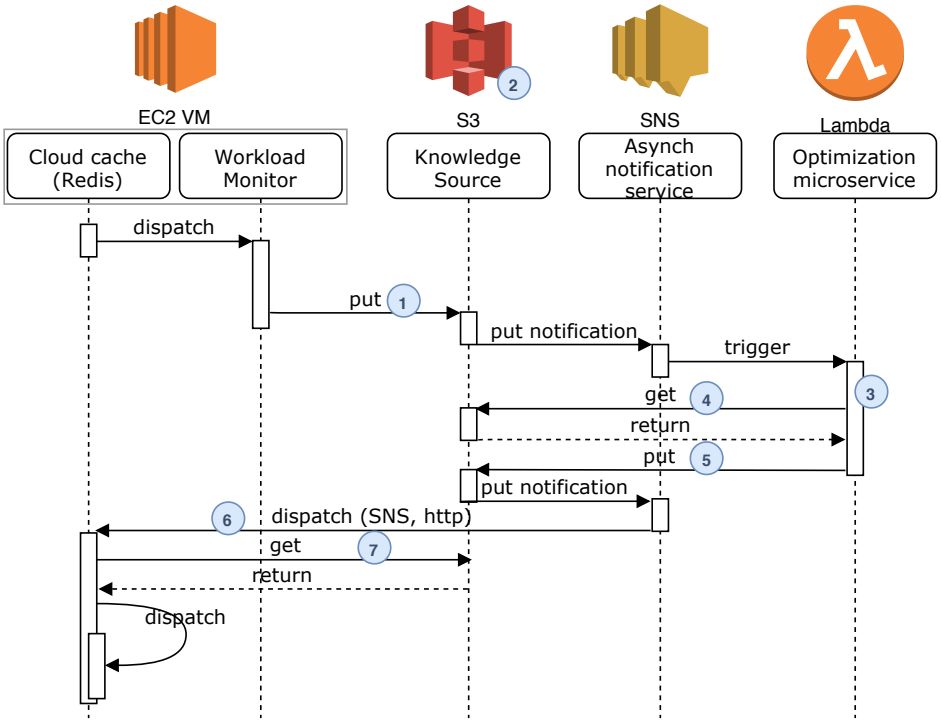

Figure 2. Components of Self-Partitioning REDiS (SPREDS) and their interactions. Table 4 describes the Amazon Web Services (AWS) services used. The circled numbers in the diagram identify the elements for which AWS charges fees.

Table 4. Amazon Web Services (AWS) products used in the SPREDS implementation.

Service Name and Description	Use in SPREDS
Elastic Compute Cloud (EC2): Server provisioning	Node hosting cache and monitoring agent
Simple Storage Service (S3): Serverless object storage	Store the monitoring data and adaptation plan
Lambda: Serverless computing (FaaS)	Serverless adaptation microservice (solver)
Simple Notification Service (SNS): Pub-sub messaging	Delivers notifications when calculate new adaptation and execute adaptation events are triggered

4.1. Summary of How SPREDS Works and Outline for the Remainder of the Section

SPREDS captures a very small percentage of the requests it receives. The sampled requests are sent in real time to the workload monitor (Section 4.2). The monitor is an external process running on the cache node; it temporarily stores the samples until a configurable number of N samples is received. The monitor then stores the N samples remotely on a S3 bucket. This triggers an SNS event that launches an AWS Lambda function to run the solver, which in turn finds the optimal partitioning of the cache using one of the two solvers (Section 4.3). Based on the solution to the optimization problem, the Lambda function generates an adaptation plan to implement the partitioning. This plan is the output of the Lambda function, which it stores on an S3 bucket. Adding a new file to the S3 bucket triggers an *execute adaptation* event, which is delivered asynchronously to the cache node using a Simple Notification Service (SNS) notification. When the cache receives the adaptation plan, it reconfigures itself according to the optimally calculated cache partitions (Section 4.4). This loop runs continuously so that the cache can adapt to changes in workload or application demand.

4.2. Workload Monitor

In earlier work, we instrumented Memcached to construct online miss rate curves (MRCs) [16]. We adopted this same approach in SPREDS, with minor changes due to the difference in internal

implementation of Redis versus Memcached. To the best of our knowledge, this is the first implementation of the SHARDS [17] MRC estimation algorithm on Redis. Next, we discuss some of the main challenges in the implementation of the lightweight monitoring approach used in SPREDS.

In computing, caches are used to accelerate access to objects stored in slower memory tiers. For this reason, caches are designed to be extremely fast when answering requests. It is important that any new functionality added to the caching software has minimal or no latency overhead. In our implementation, we move the monitoring of each request out of the critical path by implementing the monitoring agent as an external process and communicate using a high-performant asynchronous inter-process communication library (ZeroMQ, https://zeromq.org).

Another challenge is the trade-off between the sampling frequency and the accuracy of the metrics obtained. We use uniform random spatial sampling [12,17,41] to keep track of caching metrics with a low overhead. A function of the hash value of the object determines whether the object should be monitored or not, ensuring that all accesses to the same object are always monitored but only accesses to a small subset of the objects are tracked. The resulting reference stream is a scaled down representative and statistically self-similar version of the original reference stream [17,41,42]. Cache metrics can then be scaled up to approximate the metrics of the full reference stream [17,41]. The overhead introduced by the sampling method is very low for key-value stores, as these already hash the object keys before inserting them. This hashing operation can be used to implement a sampling filter [17] with sampling rate $R = T/P$ using the following operation:

$$hash(key) \bmod P < T, \tag{7}$$

where T and P are configurable parameters and *hash* is the hashing function used by the key-value store to locate an object based on its user-define *key*. A reference to an object in the cache is sent to the monitoring function if and only if it satisfies the condition in (7). Each object that passes the sampling filter condition is monitored, and it represents P/T objects in the original object request stream. To generate accurate estimation of the miss rate curves (MRCs) based on this small sample, we use the SHARDS algorithm [17]. As reported by Waldspurger et al. [17], SHARDS can process up to 17M requests per second and build MRCs that are accurate within 2.6% of the original MRC, with a constant memory footprint. SPREDS generates one MRC curve for every cache partition (each of which corresponds to a workload or application).

4.3. Optimization (Solver)

We use two AWS Lambda functions, one for each of the solvers in SPREDS. AWS Lambda is a Function-as-a-Service offering from Amazon Web Services that lets users run on-demand cloud functions that can be directly invoked or triggered by events such as adding a new file to S3. AWS Lambda charges per function invocation and as a result is typically cheaper than paying for an always-on service running on a dedicated virtual machine or container.

The first Lambda function finds a solution to the optimization problem using Algorithm 1. The second function finds a solution to the problem using a local search (hill-climbing) algorithm. We use the following heuristic to chose between the two solvers: The genetic algorithm is used when the problem is small ($\leq$7 applications sharing the cache); otherwise, we use the hill-climbing solver. We base this heuristic on observations made on exploratory experimental results (see Figure 3): The performance of the genetic solver degrades more rapidly than that of the hill-climbing algorithm, as the complexity of the optimization increases. The workloads for each partition were chosen by randomly selecting sub-traces from the Yahoo workload; experiments with other traces yielded similar results.

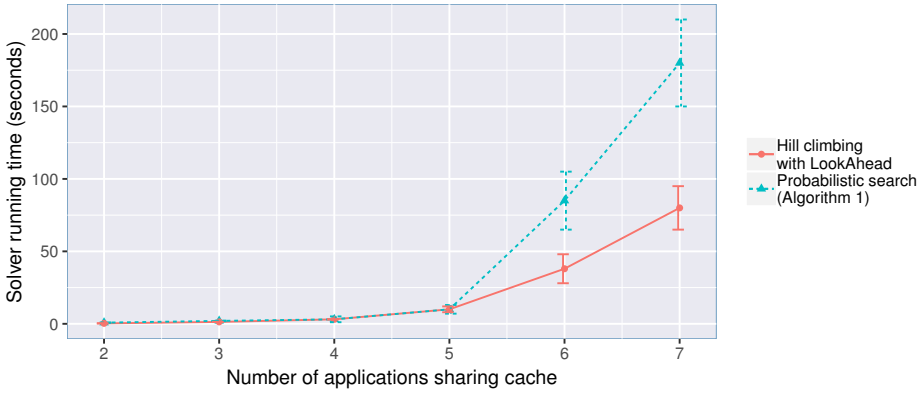

Figure 3. Average running time of the two solvers for a varying number of applications sharing the cache (partitions). Error bars show one standard deviation from the average.

4.4. Adaptation Plan and Execution

The adaptation plan instructs how to repartition the cache according to the new solution to the optimization problem. Redis supports online manual repartitioning through CONFIG SET commands, which can be communicated to the cache instances through an HTTP connection. We leverage these commands and express the adaptation plan as a set of CONFIG SET commands that change the partition sizes to match the solution found by the solver.

5. Experimental Validation and Cost Analysis

As the performance improvement due to re-partitioning the cache based on workload and application demand has already been demonstrated in prior studies [9–14,25,27], we concentrate on the specifics of the design proposed in this paper. Concretely, we conducted experiments to (1) confirm that the overhead introduced by the monitoring function is small, (2) demonstrate the usefulness in a realistic scenario, and (3) quantify the cost savings that can be achieved due to the serverless architecture proposed in this paper.

To study the performance overhead of our monitoring approach, we used a local testbed with one server and two virtual machines, one for the cache and one for the workload generator. We also ran a set of cloud (AWS) experiments in which we observe how SPREDS can self-tune and repartition the cache memory seeking to maximize the overall system utility. For these sets of experiments, we had two workload generation instances—one for each workload sharing the cache—and a third instance with the cache. In addition, the experiment orchestration scripts were run on a separate micro instance. The details of the machine characteristics and software used are presented in Table 5. The storage backends were simulated by introducing an exponentially distributed latency overhead on a cache miss.

Table 5. Description of the testbeds used in our experiments.

Type	Machine Characteristics	Software
Local	1 server (8 cores, hyperthreading, 8 GB RAM)	Ubuntu 16.04 LTS, VirtualBox (2 VMs)
	VM1 (4 vCPUs, 4 GB RAM)	Ubuntu 16.04 LTS, YCSB
	VM2 (4 vCPUs, 4 GB RAM)	Ubuntu 16.04 LTS, SPREDS/Redis v4
AWS	t2.medium instance (2 vCPUs, 4 GB RAM)	Ubuntu 16.04 LTS, YCSB
	t2.medium instance (2 vCPUs, 4 GB RAM)	Ubuntu 16.04 LTS, KV-replay [23]
	t2.xlarge instance (4 vCPUs, 16 GB RAM)	Ubuntu 16.04 LTS, SPREDS/Redis v4
	t2.micro instance (1 vCPU, 1 GB RAM)	Ubuntu 16.04 LTS, In-house orchestration scripts

5.1. Workloads

We used one application workload for the performance overhead experiments (Section 5.2) and two different application workloads sharing the same cache node for the usefulness tests (Section 5.3). The first workload—used in both sets of experiments—is a classical skewed-popularity workload in which the popularity of keys follows a Zipf distribution. Accesses are generated using the Independent Reference Model (IRM) [22] by sampling from the popularity distribution. The requests are issued using the Yahoo Cloud Serving Benchmark (YCSB) [43] for cloud and key-value storage systems, configured with the workload C of this benchmark. For the second workload, we used KV-replay [23] to replay an access trace from a large Hadoop Distributed File System (HDFS) deployment at Yahoo [44]. HDFS is a distributed file system commonly used when implementing a data lake for analytics of massive datasets. For more details on these workloads, we refer the reader to [23,43,44].

5.2. Performance Overhead

To quantify the overhead introduced by the monitoring mechanism, we ran experiments measuring the client-side latency for vanilla Redis and for SPREDS with monitoring but without the self-tuning feedback loop. This latency is the time it takes from when a client issues a request until it gets a response back from the cache. The results are shown in Figure 4a. The performance impact introduced by the monitoring mechanism is small, with a median latency that is 15% slower when the workload is being monitored. However, once this information is used to self-partition the cache, the performance overhead disappears—SPREDS is faster than Redis with static partitioning—as it is offset by the benefits in performance due to the optimized partitioning mechanism (see Figure 4b).

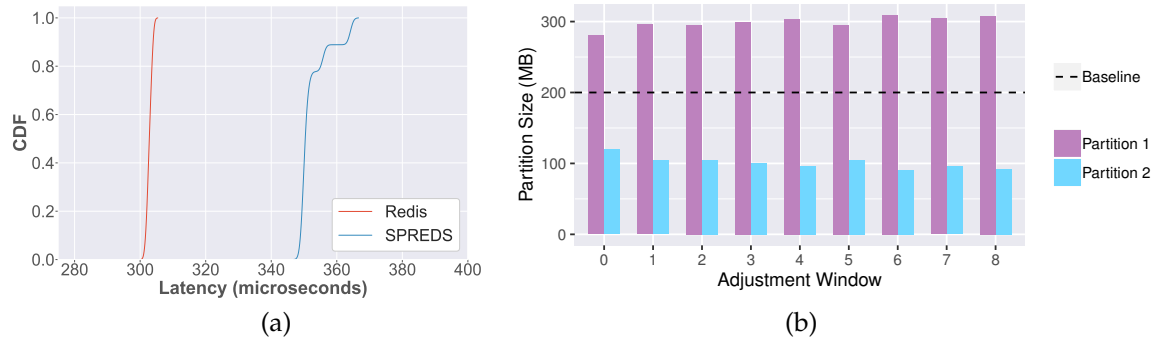

Figure 4. (**a**) Request latency for Redis and SPREDS (with monitoring but without self-tuning). The monitoring in SPREDS introduces a 15% overhead (median latency); however, this overhead becomes negligible once the system self-tunes. (**b**) Ten-hour sample run of SPREDS with two application workloads (and corresponding cache partitions).

Additional to the overhead introduced by the monitoring mechanism, there is an overhead that results from storing and reading data from the knowledge source (S3) and from the asynchronous event notifications used in SPREDS (SNS); these steps can be observed in Figure 2. We measured this overhead in a fully functional SPREDS implementation (same experiments used in Section 5.3) and found that they average ∼1800 ms per adaptation cycle versus ∼100 ms for the case of an implementation using an external always-on controller. This slowdown is big (9×) but does not affect overall system performance as the adaptation loop is executed infrequently, with reasonable values being in the order of 1 through 48 cycles per day, so this has a low impact on the performance of the full system.

5.3. Usefulness

To demonstrate the usefulness of SPREDS, we present the results of a sample run in which SPREDS shows improved performance over naive static partitioning with Redis. For numerous other positive results of optimal cache partitioning in prior related work, see Section 2.

We ran an experiment on virtual machines using AWS EC2, with a setup in which a cache node is shared between two applications. The cache is partitioned into two Redis instances, one for each workload described earlier in this section. We ran the experiments five times and present the results of one representative run. The cache was repartitioned hourly during each ten-hour run, as given by the results of solving the optimization problem. The first application issued 675 K requests during the test, with a skewed popularity distribution (Zipf, YCSB). This application is served by cache instance P1 and a storage backend that has an exponentially distributed request latency ($1/\lambda = 200$ ms). The second application issued 61.2 K requests during the test, as replayed from the Yahoo trace. This application is served by cache instance P2 and a storage backend with exponentially distributed latency ($1/\lambda = 800$ ms). At the beginning of the experiment, P1 and P2 are statically assigned 200 MB. Figure 4b shows how the size of the partitions are adjusted every hour. Table 6 quantifies the results with several performance metrics. We can observe that the hit rate of the cache that is accessed more frequently (P1) increases from 54.1% to 61.7%: a 1.42x speedup. This comes at a cost of a lesser decrease in hit rate for P2 (38.6% to 35.2%) and a corresponding slowdown (0.9). As the relative improvement perceived by P1 is greater than the penalty suffered by P2, and as P1 is more frequently accessed than P2, the overall utility improves by 7%.

Table 6. Performance improvement of SPREDS versus Redis after a ten-hour run with 9 self-partitioning cycles, as shown in Figure 4b.

Metric	Redis	SPREDS
Hit Rate, P1	54.1%	61.7%
Hit Rate, P2	38.6%	35.2%
Effective Access Time, P1	21.6 ms	15.2 ms
Effective Access Time, P2	302.1 ms	335.9 ms
Overall Utility (weighted average of EATs)	44.9 ms	41.9 ms

5.4. Costs

We calculate the cost of implementing the autonomic controller as an on-demand serverless computing service. The circled numbers in Figure 2 show the interactions and components for which AWS charges. The costs of running the cache itself are not included in the calculations; these costs are the same for all implementations. The monitor runs on the same machine as the cache and does not lead to additional charges. Table 7 details what these items represent, as well as the associated costs. The estimate given for the knowledge storage in S3 is for a six-month data retention policy. Note that some of the services listed in the table contain free service provisions: Lambda (1 million requests per month and 400,000 GB-seconds of computing time per month), SNS (1 million publishes per month, 100 thousand notifications per month, and all notifications to Lambda), and S3 (all deletes and data transfers between S3 and VMs or Lambdas in the same region). The free tier provisions that we are considering are those that are available to all AWS clients; these are unlike the EC2 free micro instances, which are only free for 12 months for new clients. (For more details, see: https://aws.amazon.com/free.)

Table 7. AWS costs for items enumerated in Figure 2 (as of December 2019, AWS us-east-2 region).

Item	Description	Cost	Quantity
1,5	S3 PUT	$0.005 per 1000 requests	Two PUTs
4,7	S3 GET	$0.0004 per 1000 requests	Two GETs
2	S3	$0.023 per GB/month	280 MB/month
3	Lambda	$0.000001667 per 100 ms; 1 GB RAM	A 1 min run
6	http SNS	$0.60 per million	1

The costs detailed in Table 7 are for one adaptation cycle. For the problem studied in this paper, the reasonable number of adaptations per day ranges from one cycle per day to 96 per day—in other words, one adaptation a day to one adaptation every 15 min.

Figure 5 shows that when the adaptation loop is executed hourly, the yearly costs of running the controller as a serverless microservice are $1.92 or 0.85% of the cost of the always-on service—if the tenant does not exceed the AWS free service provisions. If these are exceeded by other tenant activities, then the yearly cost is $10.13, or less than 5% of the cost of the always-on service. These numbers are for a one-minute optimization solver, which we calculated is enough to solve the optimizations that arise in a normal Redis deployment (for example, Redis documentation suggests 6 partitions in its tutorials). For more complex problems, the adaptation loop is likely to be run less frequently, and for many configurations expected in production, the serverless approach is expected to be better (in terms of cost) than the always-on approach.

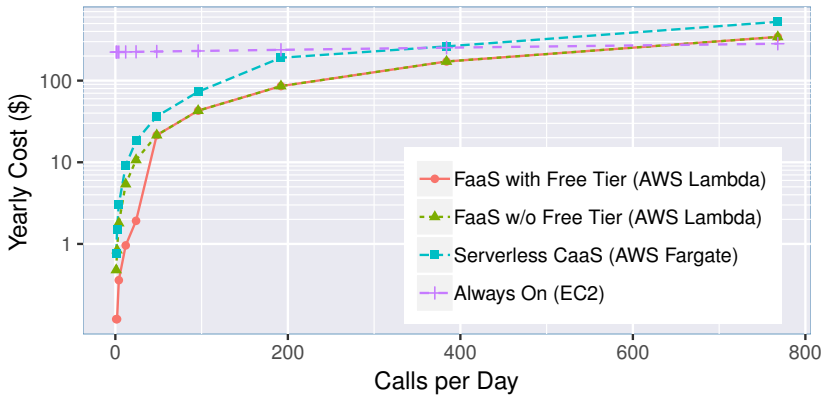

Figure 5. Costs of running the autonomic controller on AWS as an always-on service and as an on-demand serverless microservice (using Function-as-a-Service (FaaS) and Container-as-a-Service (CaaS) services). The y-axis is in log scale.

The figure also shows the costs using a serverless Container-as-a-Service (CaaS) product called AWS Fargate. We show only one cost curve for Fargate as the lines with and without free tier provisioning overlap for this service. There are no free Fargate invocations; only SNS notifications and some calls to S3 are free in this scenario. For the case of a one-minute run, using Lambda is cheaper than using Fargate. However, Fargate becomes an option when the solver running time exceeds the duration limit of the FaaS service (e.g., 15 min).

Finally, in larger organizations the solver service may be shared by multiple applications, and the always-on service may be a better alternative. For example—for the specific scenario described in this section—the always-on approach becomes a better option when the service is called 606 or more times per day (see Figure 5). At one call per hour, an organization should have at least 26 systems using the shared service for the always-on service to be the best approach. Properly choosing the deployment option—always-on serverless functions or serverless containers—is a challenge to be addressed in real installations.

6. Open Challenges

We argued for a serverless computing approach for the adaptation component of a self-optimizing system. Our proof-of-concept implementation and cost analysis show that this is feasible and useful. In this section, we identify four challenges that must be addressed for this vision to become a reality.

First, cloud functions have a maximum running time (AWS: 15min, Azure: 10min, Google: 9min). If the calculations exceed the limit, the function expires and the work is lost. While it is possible for the limit to be extended in the near future, it is nevertheless possible that some problems take too long to solve as cloud functions. For example, in 2018 AWS increased its Lambda limit from 5 to 15 min (Source: https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/). In the AWS ecosystem, the Fargate serverless container service can be used for such problems. The challenge is to design a system that can automatically use the cheapest service for the specific size (in terms of partitions) and frequency of its adaptation loop.

Second, some systems for which a serverless approach makes sense may outgrow it and require an always-on service. Again, a cloud-native solution should automatically choose the architecture that is best for the specific system configuration and complexity. This adaptation should be dynamic and transparent.

Third, we identify a challenge not specific to serverless architectures: interacting self-aware applications that use information from other systems during the adaptation decisions [45]. As the serverless approach decouples the adaptation components from the main systems, such a system could be orchestrated if we provide proper APIs and connectors.

Fourth, even though we focused on adaptation decisions (especially those related to performance tuning), the community should think about realizing the vision of fully self-aware systems [45]. Could a cloud-native architecture that leverages serverless computing offerings help in realizing such systems?

7. Concluding Remarks

The idea that complex software systems should be autonomic and self-adaptive has been around for more than a decade. Yet, while some adaptive functionality has made it to commercial systems, most real databases still depend on manual (expert) tuning. One reason self-adaptation mechanisms have not made it into production is that their tuning tends to require exploring a large configuration space or running expensive algorithms—prohibitive operations that compete for the valuable and limited resources (CPU and memory) of the system being tuned.

A way to overcome the limitations described above is to adopt a serverless microservice design for the controller. Through SPREDS, a proof-of-concept self-partitioning cloud cache, we demonstrate that this approach is feasible, useful, and overcomes the cost and performance limitations of prior designs. We encourage the community to adopt this architectural design, so that cloud-based autonomic systems can be affordably implemented in production.

Author Contributions: Conceptualization, E.F.B., X.A., J.C., J.M., H.A., C.L.A., and A.G.A.; Methodology, E.F.B., C.L.A., and A.G.A.; Writing—review & editing, E.F.B., C.L.A., and A.G.A. All authors have read and agreed to the published version of the manuscript.

Funding: We thank Google for their generous gift (Google Faculty Research Award) and Amazon Web Services for their AWS Cloud Credits for Research program which facilitated our experiments.

Acknowledgments: Finally, we thank Luis Lucio, Gustavo Totoy, Oswaldo Bayona, Jorge Vergara, and Oscar Moreno, who helped with early versions of SPREDS.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Andrade, X.; Cedeno, J.; Boza, E.; Aragon, H.; Abad, C.; Murillo, J. Optimizing Cloud Caches For Free: A Case for Autonomic Systems with a Serverless Computing Approach. In Proceedings of the Workshop on Self-Aware Computing (SeAC), Umea, Sweden, 16–20 June 2019.

2. Abad, C.; Abad, A.; Lucio, L. Dynamic memory partitioning for cloud caches with heterogeneous backends. In Proceedings of the ACM/SPEC on International Conference on Performance Engineering (ICPE), L'Aquila, Italy, 22–26 April 2017.
3. Singla, A.; Chandrasekaran, B.; Godfrey, B.; Maggs, B. The internet at the speed of light. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, Los Angeles, CA, USA, 27–28 October 2014.
4. Thiébaut, D.; Stone, H.; Wolf, J. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers* **1992**, *41*, 665–676.
5. Lu, Y.; Saxena, A.; Abdelzaher, T. Differentiated caching services; a control-theoretical approach. In Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), Mesa, AZ, USA, 16–19 April 2001.
6. Suh, E.; Rudolph, L.; Devadas, S. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing* **2004**, *28*, 7–26.
7. Qureshi, M.K.; Patt, Y.N. Utility-Based Cache Partitioning. In Proceedings of the IEEE/ACM MICRO, Orlando, FL, USA, 9–13 December 2006.
8. Einziger, G.; Friedman, R. TinyLFU: A highly efficient cache admission policy. In Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing (PDP), Torino, Italy, 12–14 February 2014.
9. Stefanovici, I.; Thereska, E.; O'Shea, G.; others. Software-defined caching: Managing caches in multi-tenant data centers. In Proceedings of the ACM Symposium on Cloud Computing, Kohala Coast, HI, USA, 27–29 August 2015.
10. Cidon, A.; Eisenman, A.; Alizadeh, M.; Katti, S. Dynacache: Dynamic cloud caching. In Proceedings of the USENIX HotCloud, Santa Clara, CA, USA, 6–7 July 2015.
11. Hu, X.; Wang, X.; Li, Y.; others. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In Proceedings of the 2015 USENIX Annual Technical Conference, Santa Clara, CA, USA, 8–10 July 2015.
12. Cidon, A.; Eisenman, A.; Alizadeh, M.; Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA, USA, 16–18 March 2016.
13. Berger, D.; Berg, B.; Zhu, T.; Sen, S.; Harchol-Balter, M. RobinHood: Tail Latency Aware Caching. In Proceedings of the USENIX OSDI, Carlsbad, CA, USA, 8–10 October 2018.
14. Cidon, A.; Rushton, D.; Rumble, S.; Stutsman, R. Memshare: A Dynamic Multi-tenant Key-value Cache. In Proceedings of the USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 12–14 July 2017.
15. Huang, K.; Wang, K.; Zheng, D.; Zhang, X.; Yan, X. Access Adaptive and Thread-Aware Cache Partitioning in Multicore Systems. *MDPI Electron.* **2018**, *7*, 172.
16. Murillo, J.; Totoy, G.; Abad, C. Instrumenting cloud caches for online workload monitoring: The case of online miss rate curve estimation in Memcached. In Proceedings of the 16th Workshop on Adaptive and Reflective Middleware (ARM at MIDDLEWARE), Las Vegas, NV, USA, 11–15 December 2017.
17. Waldspurger, C.; Park, N.; Garthwaite, A.; Ahmad, I. Efficient MRC construction with SHARDS. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, 16–19 February 2015.
18. Ebert, C.; Gallardo, G.; Hernantes, J.; Serrano, N. DevOps. *IEEE Softw.* **2016**, *33*, 94–100.
19. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Softw.* **2016**, *33*, 42–52.
20. DB-Engines Ranking of Key-value Stores. Available online: db-engines.com/en/ranking/key-value+store (accessed on 31 December 2019).
21. Psounis, K.; Prabhakar, B. A randomized web-cache replacement scheme. In Proceedings of the Joint Conference of the IEEE Computer and Communications Society, Anchorage, AK, USA, 22–26 April 2001.
22. Abad, C.; Yuan, M.; Cai, C.; Lu, Y.; Roberts, N.; Campbell, R. Generating request streams on Big Data using clustered renewal processes. *Performance Evaluation* **2013**, *70*, 704–719.
23. Boza, E.; San-Lucas, C.; Abad, C. Benchmarking key-value stores via trace replay. In Proceedings of the 2017 IEEE International Conference on Cloud Engineering (IC2E), Vancouver, BC, Canada, 4–7 April 2017.
24. Pu, Q.; Li, H.; Zaharia, M.; Ghodsi, A.; Stoica, I. FairRide: Near-optimal, fair cache sharing. In Proceedings of the USENIX NSDI, Santa Clara, CA, USA, 16–18 March 2016.

25. Van Aken, D.; Pavlo, A.; others. Automatic Database Management System Tuning Through Large-scale Machine Learning. In Proceedings of the ACM SIGMOD, Chicago, IL, USA, 14–19 May 2017.
26. Khazaei, H.; Ghanbari, A.; Litoiu, M. Adaptation as a service. In Proceedings of the ACM CASCON, Markham, Ontario, Canada, 29–31 October 2018.
27. Das, S.; Grbic, M.; others. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In Proceedings of the ACM SIGMOD, Amsterdam, The Netherlands, June 30 - July 5, 2019.
28. Van Eyk, E.; Grohmann, J.; Eismann, S.; Bauer, A.; Versluis, L.; Toader, L.; Schmitt, N.; Herbst, N.; Abad, C.; Iosup, A. The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms. *IEEE Internet Computing* **2019**, *23*, 7–18.
29. Ghahremani, S.; others. Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures. In Proceedings of the IEEE International Conference on Autonomic Computing (ICAC), Columbus, OH, USA, 17–21 July 2017.
30. Jia, Y.; Shao, Z.; Chen, F. SlimCache: Exploiting Data Compression Opportunities in Flash-Based Key-Value Caching. In Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, 25–28 September 2018.
31. Yu, Y.; w. wang.; Huang, R.; Zhang, J.; Letaief, K. Achieving Load-Balanced, Redundancy-Free Cluster Caching with Selective Partition. *IEEE Transactions on Parallel and Distributed Systems* **2019**, *31*, 439–454.
32. Xu, C.; Rajamani, K.; Ferreira, A.; Felter, W.; Rubio, J.; Li, Y. dCat: Dynamic Cache Management for Efficient, Performance-Sensitive Infrastructure-as-a-Service. In Proceedings of the European Conference on Computer Systems (EuroSys), Porto, Portugal, 23–26 April 2018.
33. Pan, C.; Luo, Y.; Wang, X.; Wang, Z. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In Proceedings of the ACM Symposium on Cloud Computing, Santa Clara, CA, USA, 20–23 November 2019.
34. Anadiotis, G. ZDNet Article — A rock and a hard place: Between ScyllaDB and Cassandra, 2017. Available online: https://www.zdnet.com/article/a-rock-and-a-hard-place-between-scylladb-and-cassandra/ (accessed on 15 April 2019).
35. Idreos, S.; Dayan, N.; others. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In Proceedings of the CIDR, Asilomar, CA, USA, 13–16 January 2019.
36. Mattson, R.L.; Gecsei, J.; Slutz, D.R.; Traiger, I.L. Evaluation techniques for storage hierarchies. *IBM Systems Journal* **1970**, *9*, 78–117.
37. Stallings, W. *Operating Systems Internals And Design Principles*, 9th ed.; Pearson Education: Upper Saddle River, NJ, USA, 2018.
38. Prügel-Bennett, A. When a genetic algorithm outperforms hill-climbing. *Theoretical Computer Science* **2004**, *320*, 135–153.
39. IBM. An architectural blueprint for autonomic computing. *IBM White Pap.* **2006**, *31*, 1–6.
40. Sidecar pattern, 2017. Microsoft Azure online documentation. Available online: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar (accessed on 13 February 2019).
41. Waldspurger, C.; Saemundsson, T.; Ahmad.; Park. Cache Modeling and Optimization using Miniature Simulations. In Proceedings of the USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 12–14 July 2017.
42. Beckmann, N.; Sanchez, D. Talus: A simple way to remove cliffs in cache performance. In Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture, Burlingame, CA, USA, 7–11 February 2015.
43. Cooper, B.; Silberstein, A.; Tam, E.; Ramakrishnan.; Sears. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, IN, USA, 10–11 June 2010.
44. Abad, C.; Roberts, N.; Lu, Y.; Campbell, R. A storage-centric analysis of MapReduce workloads. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), La Jolla, CA, USA, 4–6 November 2012.
45. Kounev, S.; Lewis, P.; Bellman, K.; Bencomo.; others. The notion of self-aware computing. In *Self-Aware Computing Systems*; Springer: Berlin, Germany, 2017.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).