Distributed Virtual Private Disk for Cloud Computing

Dottorato di Ricerca in Scienza e Alta Tecnologia, indirizzo in Informatica, ciclo XXIII

Matteo Zola
zola@di.unito.it
Dipartimento di Informatica, Università di Torino
Relatore: Prof. Cosimo Anglano

Contents

List of Figures

List of Tables

Abstract

In recent years a new technological paradigm has emerged in the computer science research community. This paradigm is referred to as Cloud Computing, a new computing paradigm whose aims are to allow users to utilize computing infrastructure over the network, supplied as an on-demand service. There are already many commercial solutions based on this approach, as well as many academic research projects that are especially focused on resource management of an infrastructure that comprises computational power, storage space and communication networks.

In particular, the demand of storage capacity is increasing, determined by, for example, scientific (e.g. physics experiments) and commercial (e.g. e-commerce sites, search engines) applications, that produce huge quantity of data. Building a data storage service that is pervasive, available, scalable and that can handle massive quantities of data has always been a priority for every distributed system paradigm and infrastructure. Cloud Computing has different goals and characteristics and thus poses new challenges in this area of research.

Cloud Computing applications, especially data intensive ones, typically need storage services that provide large amount of storage capacity and the ability of retrieving stored data at any time and in any condition (e.g., infrastructure component failures). Moreover, services are hosted and data are located on third-party resources, a condition that poses a threat to data confidentiality. In many cases, applications or system-specific functions require the availability of raw block devices, for instance when a specific file system is necessary, or when the application needs to directly access physical storage (e.g., in the case of a DBMS).

In this Thesis we propose ENIGMA, a distributed infrastructure that provides virtual disks by abstracting the storage resources provided by a set of physical nodes and exposing to Cloud Computing users, applications, and Virtual Machines a set of virtual block storage devices, that can be used exactly as standard physical disks. The important aspect of the work in this thesis is that we would like to make maximum use of resources while

providing each virtual machine the illusion of using a single resource. The system is split in two levels; at the logical level every virtual machine sees its disks as if they were a single local resource, whereas at the data level disks are distributed across the infrastructure. ENIGMA is designed to provide large storage capacity, high availability, strong confidentiality, and data access performance comparable to that of traditional storage virtualization solutions. To achieve all these design goals, ENIGMA exploits erasure-coding techniques, whereby each sector of a virtual disk is encoded as a set of n fragments, that are independently stored on a set of physical storage nodes, k of which ($k \leq n$) are sufficient to reconstruct that sector. The thesis goals are to present the ENIGMA architecture and show how the coding of sectors of a virtual disk ensures high availability in spite of failure of individual storage nodes as well as confidentiality in face of several types of attacks. Since the architecture is designed to allow ENIGMA to operate on production infrastructures and networks, we studied techniques to optimize performance. We use caching and prefetching mechanisms applied to ENIGMA in order to increase throughput and decrease average retrieval time of sectors. The performance metrics are obtained by studying ENIGMA with simulation techniques.

Chapter 1

Introduction

In recent years a new technological paradigm has emerged in the computer science research community. This trend is referred to as *Cloud Computing*, a term that relates to past concepts while introduces new advantages and new research challenges. The definition of Cloud Computing is yet to be found, and each one that is proposed it not broadly accepted. According to a recent ontology [70] *"Cloud computing can be considered a new computing paradigm that allows users to temporarily utilize computing infrastructure over the network, supplied as a service by the cloud-provider"*. In [32], the authors give the following definition *"A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet"*. Another definition is given by NIST [71] *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"*.

There are already many commercial solutions for Cloud Computing, as well as many academic research projects. All of these provide computational power as a service to the customers for running their applications. The first company that has proposed solutions based on the Cloud technology is Amazon [3] with its Amazon Elastic Compute Cloud (EC2); other examples of Cloud system are: Eucalyptus [54], Microsoft's Windows Azure [15] and Google App Engine [7], to name a few. In practice the infrastructure (computational resources connected via a network) is made available to the users for executing applications that are no longer provided by the service provider but rather it is the infrastructure itself that is made available, via an intermediate software layer, to execute users' software.

Cloud Computing provides several features, that makes it attractive and popular: is a low cost solution since it provides a pay-as-you-go pricing model, computational power is increased on-demand if users need more resources and it is highly scalable (resources can be easily added).

There are aspects that have to be solved for harnessing the full potential of the Cloud infrastructure, one of the most important is resource management. As in standard distributed computing, resources can be software or hardware, especially computational, storage and communication. Historically scientific (e.g., physics experiments) and commercial (e.g., e-commerce site, search engine) applications started to produce huge amount of data and determined an increasing demand of storage capacity. For this reason building a data storage service that is pervasive, available, scalable and that can handle massive quantities of data has always been a priority for every distributed system paradigm and infrastructure. Much effort has been done in other branches of distributed computing to deal with these issues. Given the novelty of the approach, proposed solutions may not be well suited for the new Cloud computing paradigm and should be tuned or completely changed in order to obtain the same results in the new scenario. Moreover new challenges regarding data storage requirement may arise for the new Cloud paradigm that were not previously considered.

1.1 Cloud Computing

Cloud computing is a term that hides a variety of different solutions and techniques. We will briefly sketch the main technologies behind a typical Cloud infrastructure.

The increasing availability of hardware resources (both computational and storage) at low cost and the high speed of modern network lead the companies that own large datacenters, to lease their resources to the users in an on demand fashion.

A way to provide the execution of applications over an infrastructure is through virtualization technology. It provides execution of virtual machines that run over a virtualized hardware of a real machine. A Virtual Machine (VM) is a replication of a computer system that is isolated and provides all the facilities of a real computer, like the operating system and all the environment and libraries needed by the applications. Virtualization enables us to create new instances of virtual machines, stop them at will and specify virtual hardware characteristics used by each instance based on the computational resources that underlying real hardware can provide. One of the key features of virtualization is migration. It is a facility that allows the

transfer of VM instances across distinct physical hosts. By using this tool, it is possible to move a VM to another host for increasing VM performances or tolerate failures of components. Moreover it is possible to rearrange the assignment of VM to hosts in the infrastructure, in order to maximize the resources utilization. Several solutions that offer virtualization technology are available, such as: Xen [22], KVM [9] and VMware [14].

Besides virtualization, there are other means for decoupling applications from hardware components. Remote storage access protocol, like iSCSI or IDE over Ethernet, allows an operating system (also a VM) to connect to a remote drive and provides access to it as if it were a local device. The device can be accessed from everywhere within the network, in a way that is independent from the location of the operating system.

A Cloud is a complex system, typically composed by a huge number of resources, thus it needs a way to take full advantage from the available power and automatically manage it. The most common way to do it is through an platform layer, whose primary duty is to automatically manage resources in order to optimize their utilization and provide the user with the illusion of a homogeneous system. Another purpose of the platform layer is to avoid the difficulties to deploy the applications into the VM and thus it could also provide the API to support directly applications directly into the Cloud (like for example Google App Engine).

From this point of view Cloud Computing can be seen as a collections of services, structured in layers. The top layer is called SaaS (Software as a Service) and offers to the users the possibility to run applications remotely. The layer just below SaaS is called IaaS (Infrastructure as a Service), that offers virtualization computers with guaranteed processing power and reserved bandwidth. Platform as a Service is similar to IaaS but it offers the possibility to specify a given platform, that is, it includes a custom software stack for the application. At the bottom layer the dSaaS (data Storage as a Service) provides storage for the consumers.

There are three types of cloud computing: public cloud, private cloud and hybrid cloud. In the public cloud the resources are provisioned over the internet from a cloud provider. The users application run inside the provider infrastructure and different users applications are mixed together. Private cloud refers to the cloud computing inside private infrastructure where the only applications are those belonging to one client. Hybrid clouds combine public and private clouds and have the additional complexity of choosing how to distribute the applications across the private and the public cloud.

A system that provides services should possess a way to establish the amount of resources that the consumer can use(and the price of utilization, in commercial systems) by the use of Service Level Agreement (SLA). SLA

is a mechanism for defining service utilization policies, stipulated between the user and the cloud provider. The SLA defines constraints over resources utilization (i.e., bandwidth utilization, application response time, throughput and so on) and metrics for calculating the cost of provisioning. Once the application is started and the system is working properly, there can be peaks in resources utilization or requests for starting a new virtual machine. In these cases there could be SLAs failures and there should be a redistribution of the Virtual Machines over the physical infrastructure in order to bring again the system to a stable state, where no SLA violations occur. Current research trend is to use migration for accomplishing the task of dynamically changing the virtual machines assignment to physical hosts, in response to changes of the workload. By using this tool, Virtual Machines can move to less loaded hosts in order to reduce the risk of violating the SLAs or maximize the resource utilization. Recent works applied these techniques also to reduce energy consumption, grouping VMs together and turning off unused hardware resources (server consolidation). An example of the utilization of SLA and SLA management can be found in [25].

1.2 The need of virtual private disks for cloud infrastructure

Cloud Computing applications, especially data intensive ones, typically need storage services that provide the illusion of infinite storage devices whose lifetime is not bound to that of the virtual machines (VMs) that use them. In many cases, applications or system-specific functions require the availability of raw block devices, for instance when a specific file system is necessary, or when the application needs to access directly to physical storage (e.g., in the case of a DBMS).

Virtualized Block Devices (VBD) systems (like the Amazon Elastic Block Store (EBS) [2], the Eucalyptus Block Storage Service (BSS), and the Virtual Block Store System (VBS) [35]) provide the abstraction of persistent off-instance block storage devices (henceforth referred to as *virtual disks*), whose lifetime is independent from the VM instances they are attached to and whose size can be dynamically extended at anytime.

In general, VBD systems are implemented by coupling a back-end system that provides physical storage (typically a Networked Attached Storage (NAS) device) with a front-end that provides mechanisms and protocols to access and manage virtual disks. NAS-based solutions are appropriate when the Cloud resources are located in a single data center, however may result

inadequate when storage resources are spread across different data centers, or when the virtual disks must be accessible from the resources of an Inter-Cloud [26] (i.e., a Cloud composed by a set of independent Clouds). As a matter of fact, in these situations, the following issues arise:

- *Data availability*: if the data center where the NAS is located becomes inaccessible (i.e., because of a network component failure), the storage devices it provides to the VBD system become unavailable to externally-located users or applications;

- *Data confidentiality*: usually NAS keep data private by means of encryption that, however, introduces potentially very high computation overheads, that may significantly adversely affect the performance of a virtual disk;

- *Remote access performance*: if a given virtual disk must be accessed from a virtual machine located outside the data center where the corresponding NAS resides, performance is usually limited by the single network path used to transport data.

1.3 Contribution of this thesis

In order to cope with the issues outlined in the previous section, in this thesis we propose *ENIGMA*, a distributed infrastructure that abstracts the storage resources provided by a set of physical nodes and exposes a set of virtual disks that can be used either directly by the individual virtual machines hosted on a Cloud infrastructure or as a back-end for VBD systems. ENIGMA is designed in such a way to provide a set of features tailored to Cloud Computing platforms, namely large storage capacity, high availability, strong confidentiality, high data access performance and the ability of tuning all these characteristics to specific needs of the user or of the application (even at run-time). As anticipated in the previous section, VBD system offers the ability to be independent from a particular file system optimization and can be used as a back end for many applications. Moreover, ENIGMA is, as far as we know, the first VBD system that incorporates privacy mechanisms and the ability to improve performance, features that are provided by the use of LT codes.

The virtual private disk will be accessed by standard interfaces, for example iSCSI or IDE over Ethernet. Every virtual machine could attach one or more disks as if they were normal devices, but a disk could only belong to a single virtual machine. The management of such an environment is done

through a middleware. The thesis goal is to study policies and algorithms that such a middleware should possess in order to make such a disk available to users.

The most important aspects of the work in this thesis is that we would like to make maximum use of resources while providing each virtual machine the illusion of using a single resource. We will stress a bit about this aspect. The system is split in two levels; at the logical level every virtual machine sees its disks as if they were a single local resource, whereas at the data level the sectors of the disks are distributed across the infrastructure.

ENIGMA achieves the goal of providing large storage capacity, high availability, strong confidentiality, and high data access performance, by exploiting Luby Transform (LT) codes [49]; in ENIGMA each sector of a virtual disk is encoded as a set of n *fragments* independently stored on a set of physical storage nodes, in such a way that a minimum number k ($k \leq n$) of fragments is required to reconstruct a given sector.

Data confidentiality is ensured by encoding each sector in such a way that the number k of fragments needed to reconstruct it is large enough, and by keeping private both the coding function and the addresses of the storage nodes where fragments are stored. Data availability is instead ensured by properly choosing the total number n of fragments per sector, and by spreading them on the storage nodes thoughtfully. Data access performance is achieved by simultaneously fetching several fragments of the same sector and many sectors at once, up to the bandwidth limits of the channel. Moreover the performance are increased by means of caching techniques and by dynamically increasing or decreasing the total amount of fragments per sector, for specific sectors.

In this thesis we define the architecture and functionality of ENIGMA and the mechanisms to provide reasonable performance. ENIGMA to the best of our knowledge, is the first virtual disk for Cloud Computing that exploits LT codes. The use of LT enables novel features:

- provable ability to ensure data confidentiality in face of various types of attacks;
- low encoding/decoding costs for on-the-fly operations;
- flexible and adaptive redundancy mechanism that provides availability and resilience in face of individual failure of storage nodes;

The thesis is structured as follows. After a discussion of related works (Chapter 2), we describe the architecture of ENIGMA and the mechanisms it employs to provide its functionality (Section 3), discussing the usage of

coding mechanisms (Chapter 3.2) in distributed storage pointing out how they differ from LT codes and how they are used in ENIGMA. We then evaluate ENIGMA's availability and performance (Chapter 4). In Chapter 5 we describe standard chaching mechanisms present in the literature and how they are applied to ENIGMA. Finally, we draw our conclusions and outline future research work (Chapter 6).

Chapter 2

Related Work

Storing large amounts of data in a secure, reliable, scalable and efficient way, has always been a key aspect of computer systems. The solution to this problem provided by a distributed data storage has been extensively studied in various areas of research.

The areas of research that are close to the approach adopted by ENIGMA are mainly two: traditional distributed system approaches and solutions based on Cloud Computing.

In the following paragraphs we will briefly review the main contribution of the distributed storage approach to the problem of storage.

At a higher level of abstraction, there is the file system. Various *Distributed File System (DFS)* have been developed through years, starting from NFS, Coda and others. The key aspect of such systems is that they provide remote access to files rather than access to blocks of data. Much research has been done over DFS in order to provide scalability, reliability and fault tolerance. Despite this effort, DFS are not suitable for a highly dynamic environment such as the one considered in this thesis.

Several approaches have been developed for providing distributed access at the block level. Commercial solutions are available, such as those based on proprietary technology for Storage Area Networks (for example IBM). Other approaches that provide similar facilities without using commercial products are those based on an approach called clustered storage. This approach is adopted by systems such as *Petal* [47], *FAB* [58], *Parallax* [66] and *Ursa Minor* [17], that provide the abstraction of virtual disks. In these systems data is divided into blocks and replicated or erasure coded in order to increase reliability and availability. Blocks of data are stored in a cluster and a standard interface is provided (for example iSCSI). These systems, unlike ENIGMA, are able to aggregate only storage resources located in the same data center and do not provide specific mechanisms able to deal with the issues arising

in geographically-distributed storage systems (as opposed to ENIGMA).

Another approach for the problem of distributed storage is the one based on peer-to-peer systems. Those systems (for example Oceanstore [1,55], Total Recall [46] and Storage@Home/Storage@Desk [23, 41])) provides persistent data stores designed to scale to billions of users. They provide a consistent, highly-available, and durable storage utility atop an infrastructure comprised of untrusted servers. In this case the system is geographically distributed, but they do not provide an interface to a standard disk. As for the clustered storage solutions they do not make considerations about performance. They assume that if the source of the request change it can look for the "best" fragment to improve performance. Finally, ENIGMA is – at the best of our knowledge – the only storage system providing virtual disks that directly incorporates confidentiality-ensuring mechanisms. Privacy-ensuring mechanisms are provided also by *Pasis* [34], a distributed storage system that uses a threshold scheme for coding (thus not using LT codes) to ensure availability and confidentiality, but without providing a disk-like interface.

In the next paragraphs we will review the Cloud Computing perspective on distributed storage. Cloud storage systems can be classified according to the interface they provide to users, applications and virtual machines.

Cloud file systems (e.g., the Google File System [37] and the Hadoop Distributed File System [68]), provides a file system interface, like traditional DFS. Those systems share the same interface of DFSs, but the data is replicated across the servers rather than available at a specific server (like DFSs do). In this case replication is used to provide fault tolerance.

Digital object stores, targeted to Cloud systems (e.g., Amazon's S3 [12], Sector [39], Comet [36], and Depot [50]) have been developed to provide Data-Storage as a Service (DaaS). Amazon was one of the first companies that provided its Daas system, Amazon S3 ([3], [30]). Systems like Amazon S3 provide a simple web service interface rather than a standard access to a block device and the storage is basically managed as a large pool of key-value objects of big size. Key-value is an interface in which generic "digital objects" (*values*) are associated with a key, and are stored and retrieved by using that key. Compared to these systems, ENIGMA is placed at a lower abstraction layer since it provides a virtual disk abstraction.

Virtualized Block Device systems (e.g., Amazon's Elastic Block Store (EBS) [2]), the Eucalyptus Block Storage Service (BSS), and the Virtual Block Store System (VBS) [35])) provide instead a standard disk interface, i.e. they expose to the operating system individual disk sectors. These systems are focused on providing higher-level functionalities like snapshotting, user authentication, storage sizing, while ENIGMA – by providing the abstraction of a virtual disk – complements them and can be used as their

back-end in place of the traditional NAS-based solutions they usually rely upon.

Commercial Cloud storage solutions are already available (for example Windows Mesh [16], box [4], dropbox [5] and Jungle Disk [8] to name a few), but they mainly provide an object based distributed store, rather than a disk abstraction; moreover they claim to use fixed rate erasure codes (such as Reed Solomon) or simple replication and not LT codes

Finally a good source for discussion and projects about research in the field of Cloud, and in particular of data service in the Cloud environment, is Vision Cloud [6].

Chapter 3

The ENIGMA system

In this chapter we will describe the ENIGMA architecture in details. In section 3.1 we introduce the general idea of the architecture with a brief explanation of the functionality and the role of each component. Successively in section 3.2 we will briefly review basic coding theory and how it is applied to ENIGMA. Finally (section 3.3) we will explain the operations of ENIGMA system as well as the mechanisms and protocols used by the system.

As already anticipated in the Introduction, ENIGMA provides the abstraction of a virtual disk by aggregating a set of geographically sparse storage resources. Virtual disks consist in a sequence of sectors, independently addressable, that are suitably encoded and stored on storage resources in order to achieve availability, confidentiality, and performance.

3.1 Architecture

The architecture of ENIGMA, schematically shown in Fig. 3.1, provides for two distinct logical entities, namely the *storage node* (that provides access to its local storage to store sector fragments) and the *proxy* (that coordinates the usage of storage nodes to provide virtualized disks). In an ENIGMA instantiation there may exist, at any given time, several proxies that use a set of storage nodes. However, a given virtual disk is handled exclusively by a single proxy. At the moment we assume that the functionalities of the proxy are guaranteed and no failure or loss of data occurs.

To ensure scalability, storage nodes are organized into a set of *clusters*, each one coordinated by a *cluster head*. The cluster head is a particular kind of storage node, choosen among the storage nodes that compose the cluster. If it fails it is replaced by another storage node choosen with standard algorithm of leader election. The proxy stores each fragment of each sector by dividing

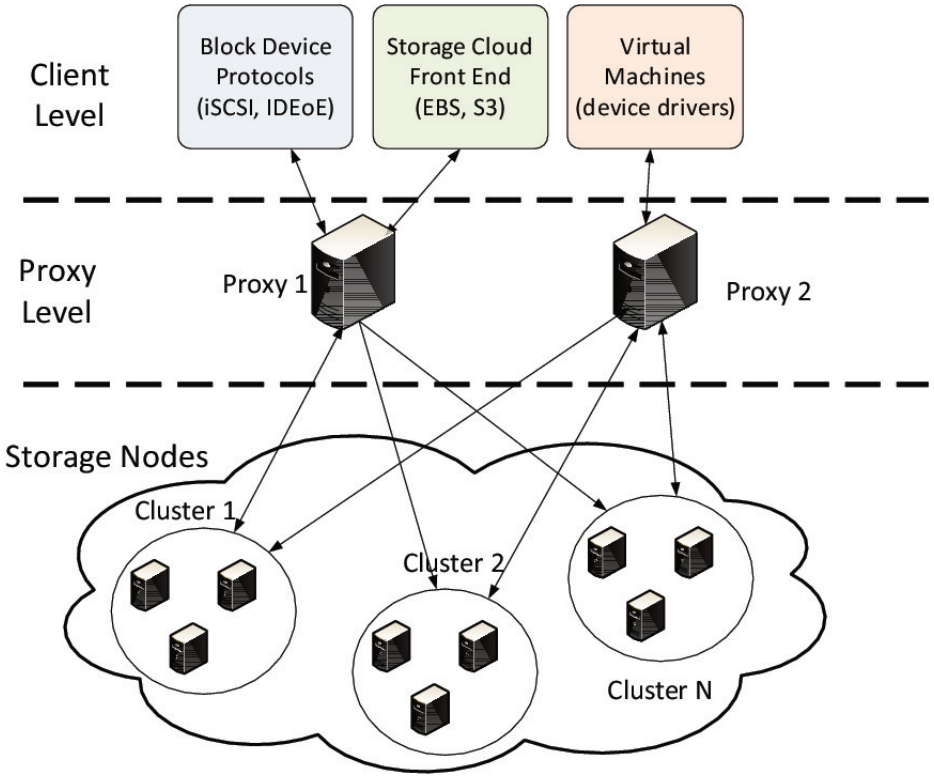

Figure 3.1: System architecture

them between two or more cluster heads (in order to avoid that the failure of a single cluster head makes the sector unavailable for a long period of time. The *client* depicted in the figure is not part of the ENIGMA architecture, but rather, uses the interfaces provided by the proxy to access the virtual disk. The client could access the disk via:

- standard Block Device Protocols: like for example iSCSI or AoE (Ata over Ethernet)
- Cloud Storage front end: Amazon clouds services are the defacto standard in this area
- device driver: they should be developed ad hoc to support ENIGMA (for either VMs or real machines)

In the next paragraphs we will briefly outline the purpose and functionality of each component of the infrastructure, then in section 3.3 we will describe the protocols and mechanisms of ENIGMA.

sector_id	**hash_id**	**cluster_list**	**CRC**	**seed**
s_1	$h(s_1)$	$ch_{11}, ch_{12}, \ldots$	CRC_1	z_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s_m	$h(s_m)$	$ch_{m1}, ch_{m2}, \ldots$	CRC_m	z_m

Table 3.1: Proxy table for virtual disk VD

3.1.1 The Proxy

The *Proxy* provides each client with access to a virtual disk that can be used by that client only, and manages the redundancy of its sectors in order to satisfy the availability, confidentiality, and performance requirements set for that disk.

All the information concerning a specific virtual disk VD is stored in a data structure called *proxy table*, whose composition and data fields are depicted in Table 3.1. In the remaining of this section we will describe the purpose of each entry in the proxy table as well as the basic operations provided by the Proxy.

In order to ensure data confidentiality, all the encoding and decoding operations for individual sectors are performed on the proxy, where the encoding seed of each sector is securely stored (we postulate that the proxy is placed on the premises of each client, that is responsible for its security). Furthermore, within the storage network, each sector s_i of a virtual disk VD is anonymously identified by means of its *hash_id* $h(VD, s_i)$, which is computed as function of the pair (VD, s_i) in such a way that it cannot be inverted to determine VD and s_i, so that an attacker who successfully decodes a sector does not know to which other sectors it relates (and thus it is not able to reconstruct the entire disk).

As it will be explained in section 3.2, each sector is encoded in fragments that are stored on two or more clusters (each cluster is a disjoint subset of the storage nodes). Upon receiving a read or write request for a given sector s_i of virtual disk VD, the proxy computes $h(s_i)$ and, by looking up the proxy table for that disk, determines the addresses of heads of the clusters $ch_{i1}, \ldots, ch_{in}$, where s_i's fragments have been stored.

The proxy forwards the read and write requests to the cluster heads in the cluster list and leave them in charge of completing fragments retrieval and fragment diffusion respectively (the complete sequence of events will be discussed in details in section 3.3).

In addition to the basic read and write operations, the proxy dynamically manages the redundancy of selected sectors, by increasing the redundancy

of the most requested sectors (in order to reduce their retrieval time) and, in a dual way, by decreasing the redundancy of sectors that are rarely used. In any case, the redundancy value of a given sector never goes below the threshold n that is set, for the whole virtual disk, to a value that guarantees a given availability level.

3.1.2 The Storage Nodes

The *Storage Nodes* are a federation of heterogeneous machines, distributed geographically and possibly belonging to different organizations.

In order to accommodate for a potentially unlimited number of storage nodes, they are organized into a hierarchical overlay network, in which nodes are grouped (as already anticipated) into clusters (each node belongs to a single cluster), each one coordinated by a cluster head that is responsible for orchestrating, across the nodes in the corresponding cluster, the execution of the operations issued by the proxy. Cluster heads form a peer-to-peer network whose topology is irrelevant for the purposes of ENIGMA (for instance, many of the overlay networks published in the literature [20] fit our needs).

Clusters and cluster heads are in charge of off-loading some of the work from the proxy, and of reducing the burden of managing the storage infrastructure. More specifically, they take care of the diffusion and retrieval of the sector fragments, and of the increase and decrease of sector redundancy, as we will discuss in section 3.3.

3.2 Coding

As anticipated in the introduction, ENIGMA features are enabled by the use of coding techniques. In this chapter we will introduce the key concept of coding theory and more specifically LT codes. Next we will describe how coding is applied to traditional distributed storage systems and afterward how LT codes are used in ENIGMA. The evaluations of the performance and feature of LT codes applied to ENIGMA are given in Chapter 4, sections 4.1 and 4.2.

3.2.1 Coding theory

Coding theory ([19]) was developed for the need of reliable transmission of information over noisy channels. In this case transmission is intended both as transmission in space (e.g. over a network) and transmission in time, by storing information on storage media (in this case coding is used to make

information available for a long period of time, copying with failures of the hard drives).

Erasure coding techniques work by splitting a given sequence of bytes into a set of independent *fragments* in such a way that, in order to reconstruct the sequence, only a subset of these fragments is required. In practice, codes split the original object in a sequence of k fragments of fixed size and encode such fragments in a longer sequence of fragments n such that any k subset of the coded fragments suffices to reconstruct the original object (Figure 3.2). Such code allows the receiver to recover up to $(n - k)$ losses in the group of n fragments. The fraction $r = k/n$ is called *code rate*; if more fragments (k') are used to reconstruct the original object, the fraction k'/k is called the *efficiency* of the code.

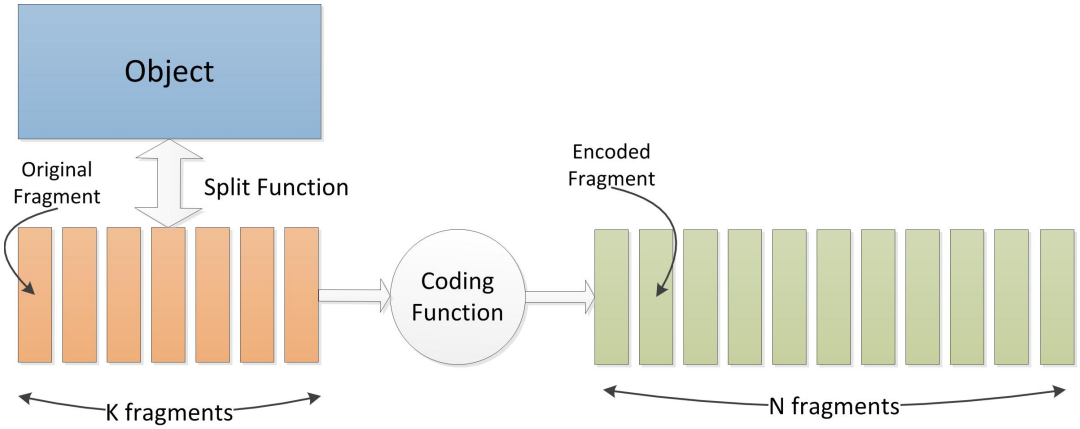

Figure 3.2: Coding

Optimal erasure codes have the property that strictly any k out of n fragments are sufficient to recover the object and that the rate r is fixed, that is, the values we choose for k and n could not be changed, otherwise we have to recode the object. Traditional optimal codes such as Reed-Solomon have been successfully applied in various contexts, but decoding time quadratic in n makes them expensive for large files (unless k is kept fixed), that is, as n grows the cost of coding introduces too much computational overhead. To cope with this issue, researchers have proposed a new class of erasure codes with faster decoding times.

Near-optimal erasure codes are a class of codes with sub-quadratic decoding times. Codes like, for example, Tornado codes [27], LT codes [49], Raptor Codes [60] and Online Codes [51], produce coded fragments that are simple xor of the original fragments in which the object is first divided. That is, the object F is composed of fragments f_1 through f_k and the coded fragment c_1

might be computed as $f_1 + f_2$, for example. The linear relationship between original fragments and encoded fragments vary with the scheme. The price we have to pay for fast decoding is that this family of codes are not optimal, that is, they require $o \cdot k$, where o is the *overhead* factor that is greater than one, but has the property to approach to one when k approaches infinity. The overhead defines the number of fragments, beyond the k, that have to be obtained to reconstruct the original object. These additional fragments, that represent the ovehead value, are called ϵ. This family of codes (except Tornado Codes) have also the property to be rateless, which means that the rate r is not fixed a priori and thus this codes could encode a given sequence of fragments into a theoretically infinite stream of coded fragments.

The idea behind this particular family of erasure codes comes from the concept of *digital fountain* [53]. Digital fountain is an abstraction of erasure coding that has properties similar to a fountain of water: "when you fill your cup from the fountain, you do not care what drops of water fall in, but only want that your cup fills enough to quench your thirst"([53]). With digital fountain a receiver obtains encoded fragments from one or more sources, and, once enough packets are obtained, the receiver can reconstruct the original object with high probability, regardless which packets has obtained and how many packets get lost in the transmission. An idealized digital fountain should be able to generate a potentially infinite number of encoded fragments and the receiver should be able to reconstruct the original object with any k encoded fragments received.

LT codes [49] belongs to the class of rateless codes (the digital fountain approach), where the encoder can generate on the fly novel coded fragments without the need to fix the rate in advance as for classical linear block codes. LT codes are asymptotically optimal, i.e. the overhead ϵ vanishes (i.e. approaches to 0) for large k, and both the encoding and decoding algorithms have a low computational cost.

In LT codes, an encoding symbol is generated by choosing a degree d according to the Robust Soliton Distribution (RSD) and then choosing d distinct symbols uniformly at random and set the encoded symbol to be exclusive-or of these d symbols. LT codes have an implicit graph structure formed by the d symbols (the neighbors) of each encoded fragment. Each encoded symbol must have associated the list of its neighbors. This is accomplished by additional information such as packet identification number used to seed a pseudorandom generator. Each implicit graph is represented by a bitmap, that is a sequence of 1s and 0s that represents the neighbors of the coding fragment. In practice, the bitmap $b_{i,l}$ is generated by choosing the degree d according to the RSD $P(d = k) = \mu(k)$, then d out of the k fragments of the sector are chosen uniformly and combined to form the coded fragment.

In practice (Figure 3.3), the bitmap could be seen as a sparse graph where each position represents one of the original fragments in which the object is first divided. If a given position of the bitmap is set to "1" its corresponding fragment is connected to the graph. The encoded fragment is the result of XORing all the connected fragments of a given graph. Each graph (and thus each encoded fragment) is generated from a sequence of random numbers starting from a common seed.

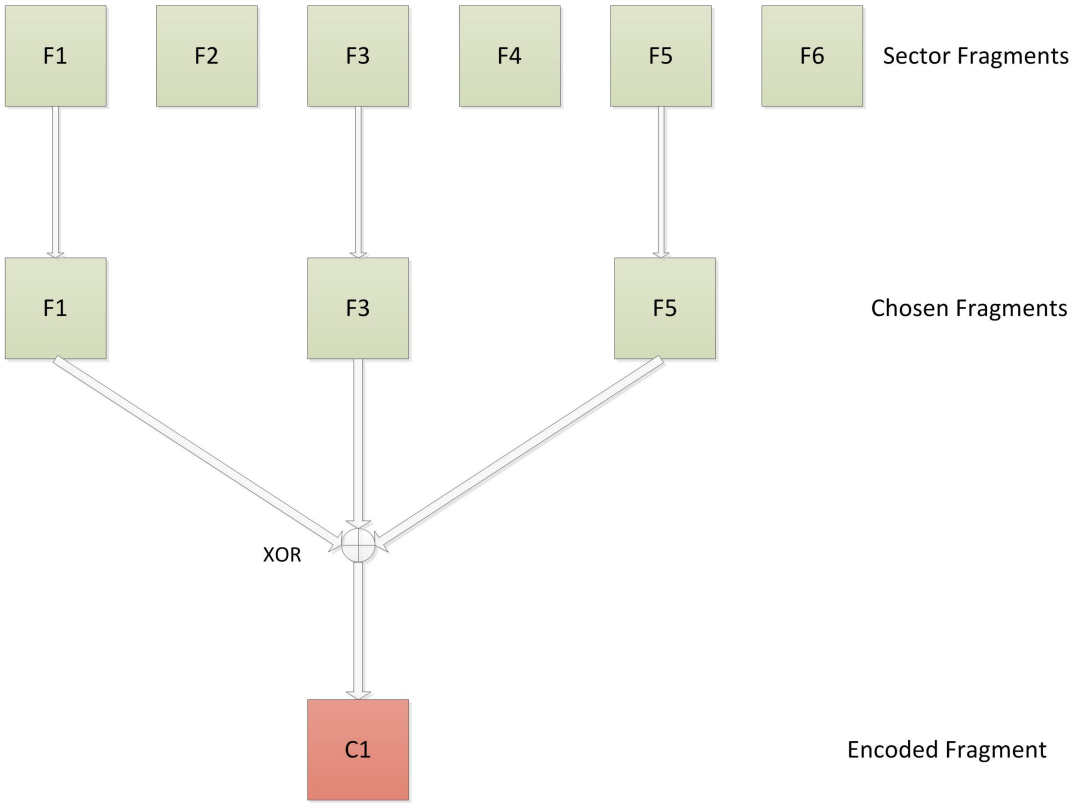

Figure 3.3: LT codes: overview

3.2.2 Coding and Distributed Storage

Storage systems have always used some form of fault tolerance. The easiest way to tolerate failures in distributed storage systems, is to replicate data and distribute them into the infrastructure. Replication works by partitioning an object into n blocks of fixed size and replicate each block (i.e. make m exact copies of the original block) with a predefined replica factor, for example replicate each block twice, with $m = 2$. This simple method provides both fault tolerance and improved performance over non-replicated systems, but at

the cost of higher storage space. A block that is replicated m time can tolerate the failure of $m-1$ of nodes holding a replica of a particular block. In the example before we doubled the storage space used to tolerate the loss of only one replica per block. Erasure coding schemes improve both fault tolerance and access performance of replicated systems. In the field of distributed system is still debated if erasure coding techniques are better than replication to provide fault tolerance and access performance ([67], [48] and [56]). In the context of ENIGMA the use of erasure codes, and in particular LT codes, is motivated by the requirement and the features that ENIGMA offers.

Traditional hard drives, for example, use Reed-Solomon for RAID systems. As discussed previously (section 3.2.1) Reed-Solomon codes do not scale well if k is high (in order to keep k low we have to increase fragment size). In [46] both replication and erasure coding are compared. Given a specified amount of redundancy and replication the authors calculate the delivered availability. As expected, for the same level of availability, an erasure-code mechanism requires a lower storage overhead compared to that needed by replication. Many other systems in the literature use erasure coding techniques ([34] and [17] for example), but they tend to use a fixed rate of redundancy for a given failure model, in order to provide the desired availability level. ENIGMA, on the other hand, increases and decreases the rate of selected sectors for both performance and availability improvement.

Surprisingly, modern Cloud Storage system, like for example Amazon S3 ([3], [30]), use simple replication instead of erasure coding. This could be explained by the computational overhead of erasure coding large files (the size of S3 objects is up to 5 $Terabytes$) and by the relative ease of managing replicas instead of erasure coded fragments.

3.2.3 Coding in ENIGMA

In ENIGMA coding is applied to the sectors of a virtual disk. Each sector is the basic object to which coding function is applied and it has its own availability level, chosen by generating a suitable amount of encoded fragments. We use LT codes to create the encoded fragments. In the following section we will explain in details how coding is applied to sectors and we will introduce a second level of coding used for improving performance.

Sector encoding

The encoding of a virtual disk VD, composed of m consecutive sectors s_i of identical size, is performed as follows. Each sector s_i is first divided in k fragments $s_i^1, \ldots, s_i^k$ and, by subsequently applying a *coding function*, it is

transformed into a sequence of n fragments $c_{i,l}$, $l = 1, 2, \ldots, n$ with $n > k$, computed as an exclusive-or (XOR) of the fragments of sector s_i (which amounts to consider linear coding in the Galois field of size 2, GF(2)) as:

$$c_{i,l} = b_{i,l} \cdot s_i = \sum_{j=1}^{k} b_{i,l}^j s_i^j, \tag{3.1}$$

where $b_{i,l} = b_{i,l}^1, \ldots, b_{i,l}^k \in \mathrm{GF}(2)^k$ is the bitmap of the encoded fragment $c_{i,l}$. In practice, the bitmap is a random vector of length k in which each element is either 0 or 1. The number of 1s of a bitmap is called *degree*. Any $K = k + \epsilon$ fragments taken from the n coded fragments can reconstruct the original sector, and thus up to $n - K$ failures of the nodes can be tolerated. The bitmaps are produced sequentially by a standard random generator initialized with a given *seed* z_i. The random outcomes leading to the bitmap $b_{i,l}$ can be reproduced by knowing z_i and the sequential number of the coded fragment l. The couple (z_i, l) can be interpreted as a key, required to correctly interpret and decode the coded fragment. On the Fly Gaussian Elimination (OFG) [24] decoding algorithm is used to decode the fragments.

Second level encoding

In order to increase the redundancy level of a given sector, its key must be used to generate other encoded fragments (in the potentially infinite sequence of fragments). If we wish to keep the encoded sector private, thus not revealing the key, we have to devise another method for the redundancy increase operation (we use coding because its computational overhead is less than that of encryption). For this purpose we use a second level of coding that combines encoded fragments to obtain new encoded fragments that could be used as normal encoded fragments. This method could be used by a third party for increasing the redundancy level of a given sector without knowing the secret key.

A third party may increase the redundancy of sector s_i by *recombining*, by means of pairwise XOR operations, a subset of fragments $c_{i,l}$ of sector s_i in its possession. The recombination of two fragments $c_{i,l1}, c_{i,l2}$ of s_i corresponds to create a new combined fragment $c_{i,l1,l2}$ by XORing them, i.e. $c_{i,l1,l2} = c_{i,l1} \oplus c_{i,l2}$.

In Figure 3.4 we can see a simple example of how this second level of coding works. We suppose that a given sector is divided into 2 fragments and encoded in 3 fragments, namely $C1$, $C2$ and $C3$. Any two out of the encoded fragment suffice in reconstructing the original sector, thus this simple coding can tolerate a single failure. If we want to increase the fault tolerance

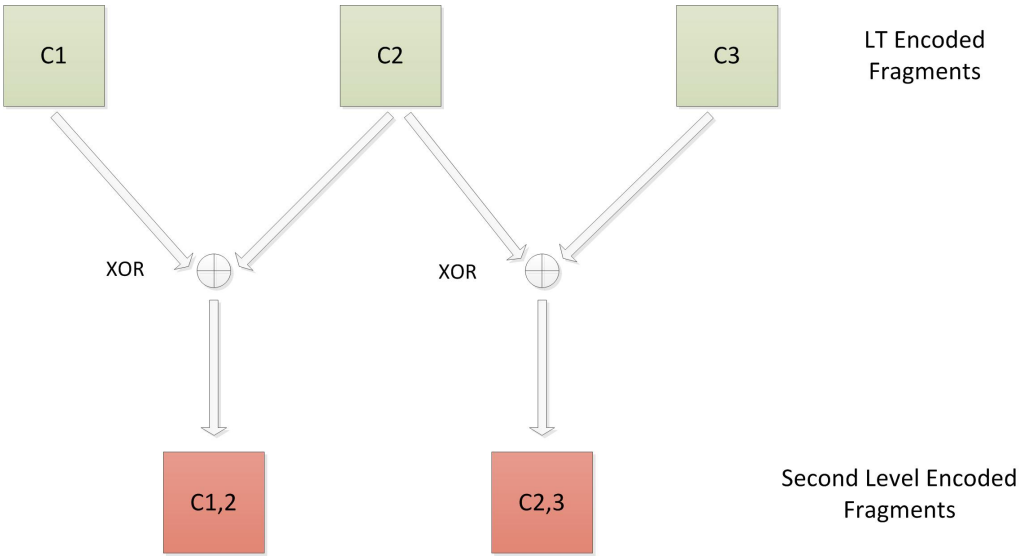

Figure 3.4: Second level of coding: example

of this configuration, we can apply the second level of coding, as explained before, and combining the fragments to create two new encoded fragments. The fragments called $C1,2$ and $C2,3$ in the figure are obtained by XORing fragments $C1$, $C2$ and $C2$, $C3$ respectively. Those fragments could be used as any other encoded fragment because the rule that any two fragments can reconstruct the original still holds. For example if we get fragments $C1$ and $C1,2$ we can XOR them and obtain fragment $C2$. Then the decoding process could continue as in the standard case.

Sector encoding (along with second level coding) provides the basic mechanism for simultaneously achieving availability, confidentiality, and performance in the following way:

- *Availability*: the availability of a virtual disk can be set to a given level by properly choosing the values of k and n (and correspondingly ϵ); furthermore, the encoding technique we use allows one to dynamically change any of these values for individual sectors, without having to do the same for all the other sectors of the disk.

- *Confidentiality*: storage nodes are unable to reconstruct any sector since (a) the coding seed z_i of a sector is unknown, and (b) far less than K coded fragments are stored per each node. Plain text attack are not feasible because data inside fragments is arbitrary. Even if the attacker can see ASCII text inside a fragment it is not sure if it is a real text or an ASCII visualization of arbitrary data. In ENIGMA,

coding provides confidentiality and not autentication or criptographic properties. It is however possible to provide such functionalities, for example with standard techniques as discussed in [29] and [65].

- *Performance*: the fragments of a given sector, being stored on different storage nodes, can be retrieved in parallel, thus reducing the time required to retrieve a sector with respect to the case in which the whole sector is stored on a single node.

Sector encoding is the basic building block of ENIGMA and it provides the functionalities to perform read and write operations of the virtual disk. This operation are provided by the architecture that will be described in the next Chapter.

3.3 Operations

As anticipated previously, this section is devoted to the explanation of the details of each operation performed by the infrastructure. In addition to the standard operation of a disk drive, namely read and write, ENIGMA infrastructure has to provide other functionality in support of the aforementioned operations.

One of the peculiarities of ENIGMA is the ability (provided by the proxy), to perform *overlapped operations*, where an operation can be performed as soon as its request arrives, without having to wait the completion of another operation that has been issued previously. This behavior (used to improve performance) is different from the one exhibited by physical hard disks, where the presence of mechanical components that may need to be repositioned each time a new sector must be accessed imposes, in practice, the completion of a pending request before a new one can be performed. The only limitation for the number of operations performed in parallel, is the amount of bandwidth that the proxy is able to provide, meaning that the number of operations times the size of the sector should not exceed the available bandwidth, in order to prevent congestion.

In the next sections we will explain each operation in details.

3.3.1 Write

When a sector s_i is written for the first time, the proxy determines the set $ch_{i1}, \ldots, ch_{in}$ of cluster heads in charge of storing its encoded fragments, and stores this information – together with $h(s_i)$ and its encoding seed z_i – into the proxy table.

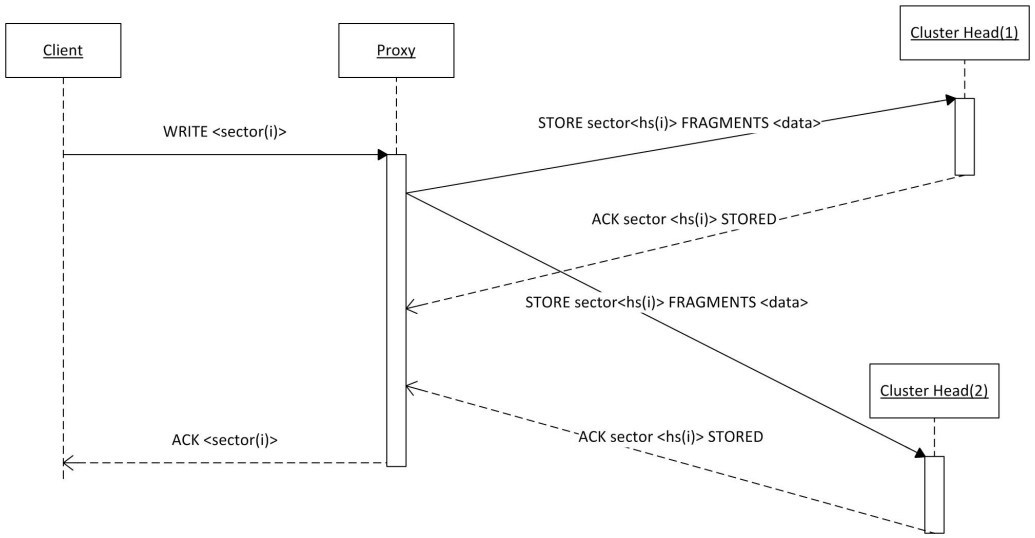

Figure 3.5: Sequence diagram for write request

The *CRC* field in the proxy table stores a cyclic redundancy code (*CRC*) used to identify possible inconsistencies in the sector data, that might be caused – for instance – by data pollution attempts. More specifically, when a sector is written, its CRC is computed and stored by the proxy before it is encoded. When a sector is read, after it has been decoded, the proxy recomputes its CRC proxy and compares it with the one stored before the write operation: if the two values coincide, then the proxy deduces that no data corruption (due to either normal events of malicious actions) occurred.

In figure 3.5, we can see the sequence diagram for the write command, involving client, proxy and cluster head (we draw two cluster head for clarity). When the proxy receives a write request for sector s_i, it encodes s_i and sends the resulting fragments to the cluster heads stored in the corresponding row of the proxy table. Each fragment $c_{i,l}$ of s_i is stored as a tuple $< h(s_i), l, c_{i,l} >$, where l represents the sequential identifier of the coded fragment and $c_{i,l}$ is the actual data; in the picture "$< data >$" represent a group of fragments that a cluster head has to store inside its cluster. The sector content can be retrieved only by the proxy from any K tuples $< h(s_i), l, c_{i,l} >$. For each fragment of s_i the proxy chooses at random, with uniform probability, two or more clusters that will store it; in this way we avoid that a failure of a single cluster head compromises the entire sector. If the sector is *empty* (that is, it has been never written since the creation of the virtual disk), then the proxy considers the write operation to be completed.

Instead, if the sector is non empty (that is, it contains data that must be overwritten as result of the write operation), then all the fragments of s_i

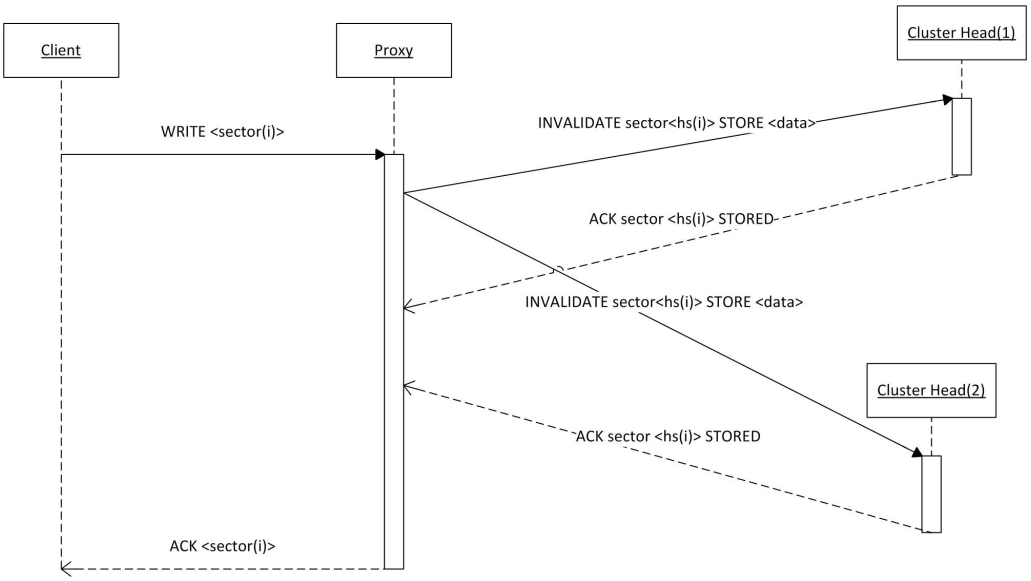

Figure 3.6: Sequence diagram for write request(invalidate command)

that are stored on the storage nodes must be overwritten before the write operation can be considered complete (3.6). If this was not done, a read operation for sector s_i overlapped with a write of the same sector could lead to consistency problems in case some of the old fragments is retrieved and combined with some new ones. We will clarify this statement with an example. Assume sector s_i has been just written and shortly after it has been requested again by the client. At this time there are two kind of fragments in the system that belong to sector s_i: the original fragments and the fragments created with the last write. The system is not able to distinguish the two kind of fragments and thus opunt the read request for sector s_i it will retrieve a mix of old and new fragments. During the encoding phase the reconstructed sector will not match the CRC stored at the proxy (infact the combination of different fragments will pollute the code, resulting in a useless sector).

To avoid this problem, the proxy keeps the just-written sector s_i into a local cache until the write operation is terminated (that is all the fragments of s_i have been committed on the respective clusters), and sends to each cluster head an *invalidate* command together with the list of fragments assigned to that cluster head. Upon receiving this message, each cluster head eliminates the old fragments, stores the new one and, when these operations have been completed for all the fragments, sends an acknowledgment back to the proxy.

In both figures 3.5 and 3.6, we depicted the course of events from the client-proxy-cluster head perspective. When a cluster head receives a group of fragments it has to store them on suitable storage nodes. This operation

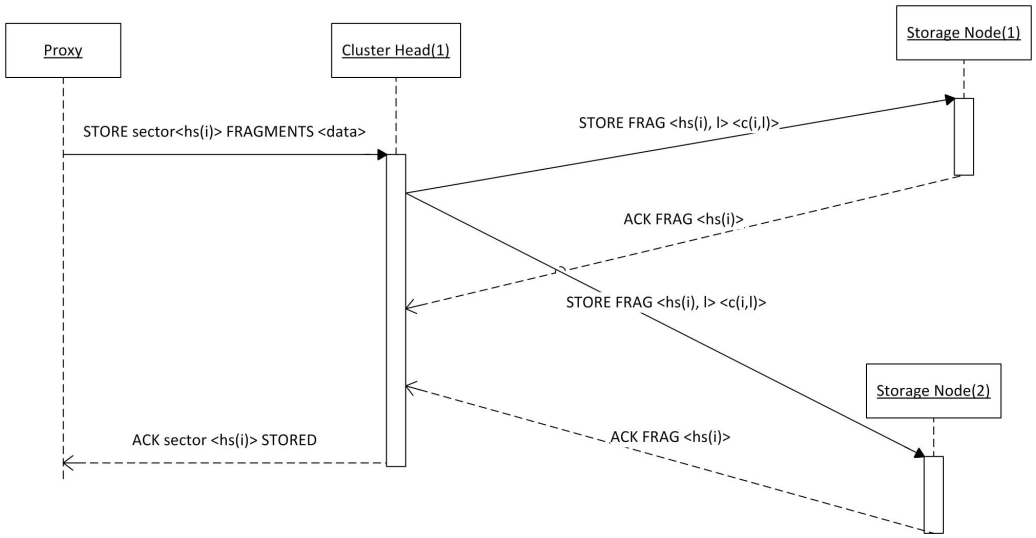

Figure 3.7: Sequence diagram for fragment diffusion

is accomplished by the fragment diffusion protocol.

Fragment diffusion

When a virtual disk VD is created, the cluster list of each sector is empty and it is filled when the sector is written for the first time. When sector s_i must be stored, the proxy sends to each one of the cluster heads responsible for s_i (see Sec. 3.1.1) some of s_i's fragments. If sector s_i is written for the first time, the cluster head places the fragments on suitable storage nodes picked up at random among the components of the cluster (Figure 3.7). Conversely, if s_i has been already written, the message sent by the proxy contains both an *invalidate* command and the new fragments to be stored in the cluster (Figure 3.8). In this case, before storing the new fragments, the old ones are deleted. After these operations have been completed, an acknowledgment is sent to the proxy that, as already discussed, can discard the locally-cached copy.

3.3.2 Read

When the proxy receives a read request for sector s_i, it sends a read request to every cluster head in charge of holding the fragments of s_i. In Figure 3.9 we can see the sequence diagram of the read protocol. The cluster head, by using the *fragment retrieval protocol* (described in the next section), forwards the request to all the storage nodes of the cluster that actually store fragments

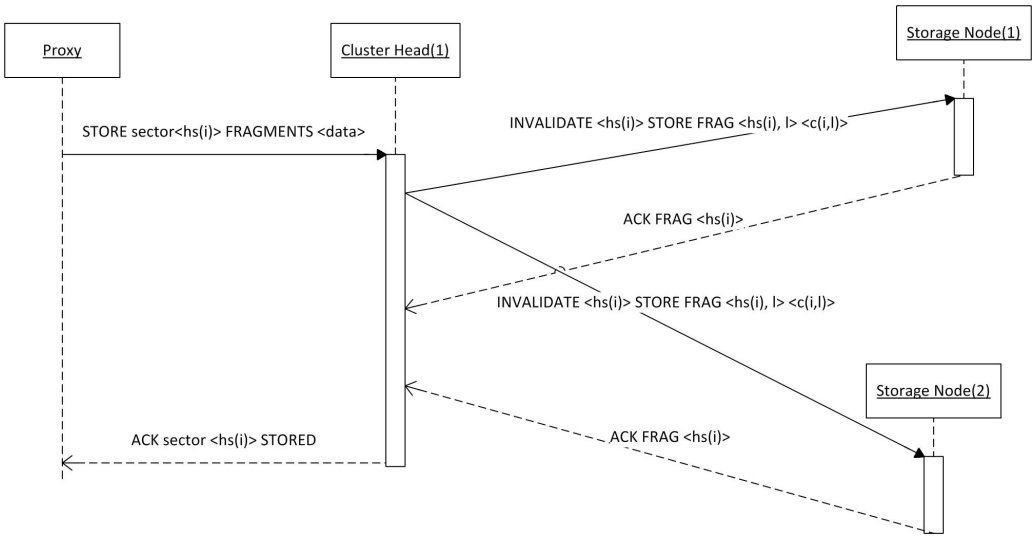

Figure 3.8: Sequence diagram for fragment diffusion(invalidate command)

associated to $h(s_i)$. In Figure 3.9 cluster is a graphical shortcut to avoid drawing each storage node, but it should be interpreted as the collection of storage nodes that form a cluster. When the proxy has received K fragments, it decodes s_i and discards all the additional fragments received after the CRC has been successfully computed. If all the N fragments of a sector are sent simultaneously, or in a short period of time, the proxy could become a bottleneck. We plan to investigate as future work whether to adopt PULL methods that request only k fragments (plus an arbitrary epsilon) to reduce the load at the proxy.

Fragment retrieval

Upon receiving a retrieval request for a specific sector s_i (i.e., for the corresponding $h(s_i)$) value, the cluster head sends a broadcast message to all the members of its cluster requesting all the fragments associated with $h(s_i)$. Each storage node in that cluster reacts to that message by directly sending to the proxy all the fragments associated to $h(s_i)$.

3.3.3 Redundancy increase

Redundancy increase is accomplished by the proxy that, after choosing which sector s_i has to undergo redundancy increase, sends an *increase* command to two of the cluster heads storing fragments of s_i. These two cluster heads combine and code the fragments in their possession, and then send the newly

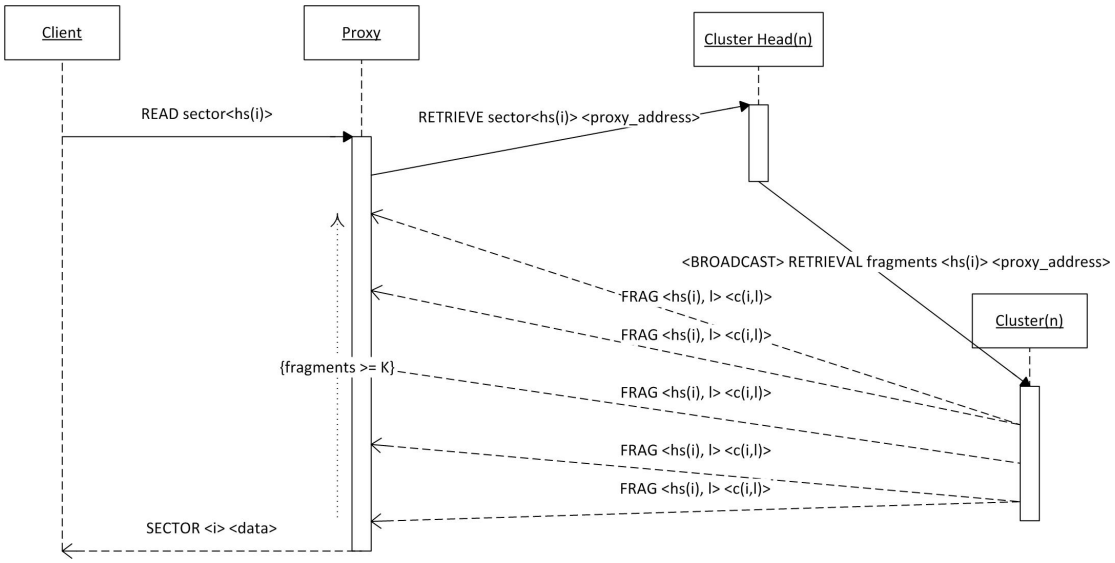

Figure 3.9: Sequence diagram for read request

generated fragments to another cluster head that will store them in its storage nodes. In this way, redundancy is increased without requiring the intervention of the proxy, that has only to update the table of the corresponding virtual disk in order to update the list of cluster heads managing s_i.

Figure 3.10 depict the redundancy increase protocol, that we will discuss in detail. When the proxy decides to increase the redundancy of sector s_i, it sends a message $increase(h(s_i)$ to two cluster heads ch_x and ch_y. These cluster heads are asked to combine their fragments associated with $h(s_i)$ and store them in a third cluster (which could also be one of the former two), with a standard write operation. Either one of ch_x and ch_y could act as a leader and start the coding protocol. As explained in Chapter 3.2 Section 3.2.3, the second level of coding enables individual nodes to recode fragments without exposing them the encoding seed z_i (to preserve confidentiality), and without requesting the intervention of the proxy (in order to avoid a potential performance bottleneck). The leading cluster head combine its fragments with that of the other cluster head the proxy has recommended. For example the combination of two fragments $c_{i,l1}, c_{i,l2}$ of s_i creates a new fragment $c_{i,l1,l2}$ that is stored as a tuple $< h(s_i), l1, l2, c_{i,l1,l2} >$. The recombined fragments can be used by the proxy to decode the sector content by XORing bitmaps corresponding to the combined $l1$ and $l2$ packets.

When the redundancy increase operation is finished, the leading cluster head sends a message to the proxy, containing the identificator of the cluster head responsible for storing the new fragments. The proxy, then, updates

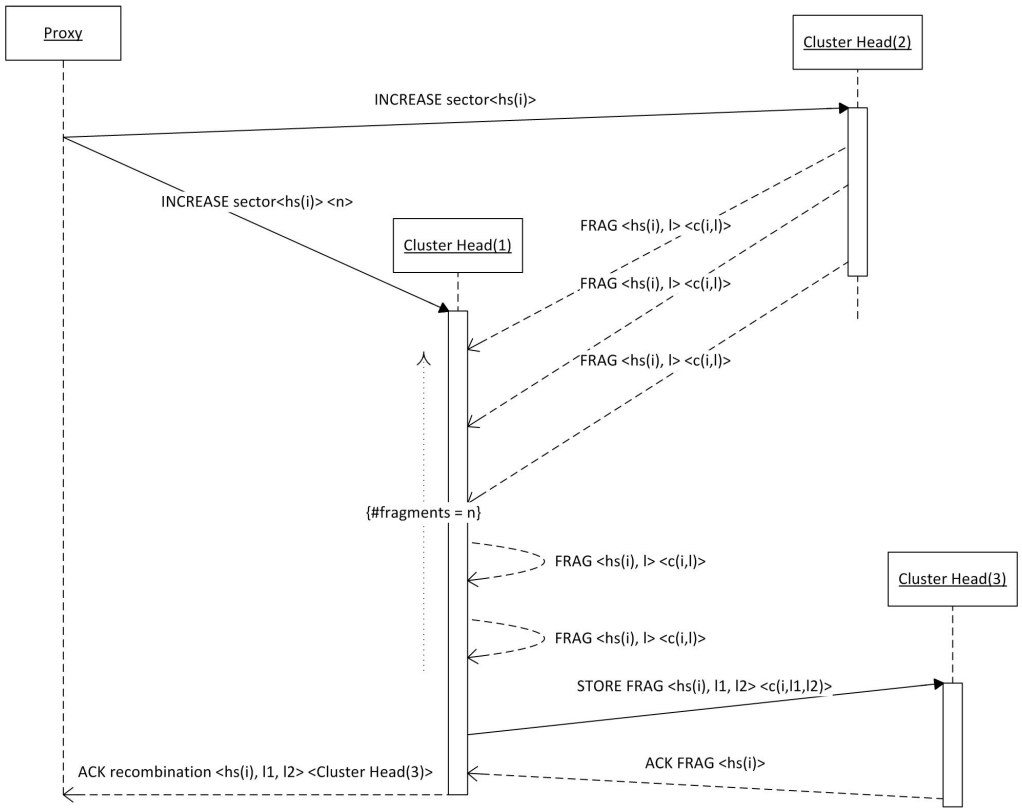

Figure 3.10: Sequence diagram for redundancy increase

the entry in the disk table for that sector.

3.3.4 Redundancy decrease

Redundancy reduction for sector s_i is instead performed by having the proxy ask, to every cluster in charge of handling fragments of s_i, to delete a given number of fragments (determined by the proxy) sufficient to bring redundancy back to its standard value n.

In Figure 3.11 we can see the redundancy decrease protocol in action. Redundancy decrease command is sent by the proxy to each cluster head in the list for that particular sector s_i. Upon receiving the command $decrease(h(s_i))$, only the cluster heads that store recombined fragments perform the operations (decreasing is done in order to avoid wastage of storage resources when further redundancy is not nedeed). They send a broadcast message to all the storage nodes in their cluster asking to delete the recombined fragments of the sector. Each cluster head that deleted some fragments send a message to the proxy that updates the disk table.

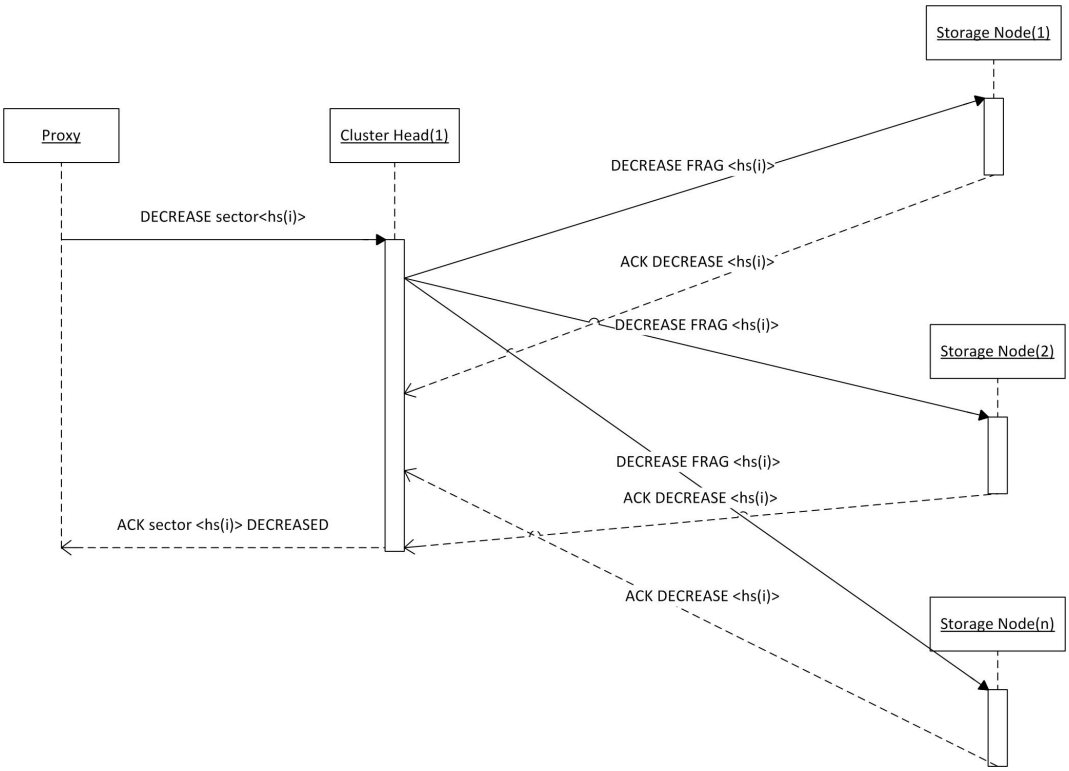

Figure 3.11: Sequence diagram for redundancy decrease

3.3.5 Maintenance operations

Another duty of the overlay of storage nodes is to periodically check the status of its participants. This could be used by the proxy as a keep alive function that is necessary in order to monitor availability of sectors. The proxy periodically sends a *keep alive* message to each cluster heads that store fragments of a particular sector and it evaluates the responses. The cluster heads that receive the keep alive message monitor in turn the storage nodes of their cluster. When storage nodes availability falls below a guaranteed threshold, the proxy can issue an increase redundancy command, restoring the original availability.

If the number of original $c_{i,l}$ fragments decreases too much, the second level of coding could not suffice to reconstruct the sector (recall that it encodes only a subset of fragments) and original redundancy has to be restored. This operation is performed every time a write occur. If write operation are scarce, the proxy, periodically, takes care of collecting the fragments recon-

structing the sector, coding it again and spreading the fragments.

In this chapter we described the ENIGMA architecture and the operation that it provides in order to give to the users a virtual disk, capable of performing the same tasks of a regular disk. In the next chapters we will describe caching techniques used to improve access performance and we will discuss the results achieved by ENIGMA.

Chapter 4

Availability assessment and performance evaluation of ENIGMA

In this chapter we evaluate the characteristics of the ENIGMA system. As pointed out in section 3.2, the use of rateless codes allows ENIGMA to provide confidentiality, availability and remote access performance. In sections 4.1 and 4.2 we provide a quantitative evaluation (theoretically and via simulation) of, respectively, confidentiality and availability. In the following sections we evaluate the performance of ENIGMA. Typically disks and distributed storage systems are evaluated with regard to two measures of performance: latency and throughput. In order to study the feasibility of our approach we opted to study ENIGMA performance with simulation techniques. In section 4.3.1 we first describe the simulation methodology used, then we study the relationship between redundancy and latency, and, in section 4.3.3 we quantitatively examine the throughput that ENIGMA could provide. In all the experiments we suppose that the nodes composing ENIGMA are perfectly reliable, in section 4.3.4 we will discuss how performance are affected by using unreliable nodes.

4.1 Data Confidentiality Assessment

In ENIGMA, data confidentiality is guaranteed by three major means, namely fragment dispersion, disk addressing policies and the linear coding of fragments. According to the distributed nature of ENIGMA, the list of clusters hosting a given sector is not available in the clear and is secured in the proxy. Therefore both cluster head and storage nodes cannot retrieve all the

coded fragments of a sector. At the addressing level each sector is identified through the anonymous *hash_id*, that is meaningful for the proxy owning the disk only. In this way cluster and storage nodes are neither able to associate a coded fragment to a given VD nor they know the sector sequential order.

It is worth pointing out that the confidentiality related to the dispersion and addressing policies is not robust to malicious nodes able to mimic the proxy requests. In fact, an attacker could conceive a malicious node implementation that sniffs the protocol signalling from a proxy, tries to reconstruct the association between the *hash_id* and (VD, s_i), then attempts to retrieve at least K coded fragments by flooding the cluster head nodes of read requests. Nonetheless, this weak levels of confidentiality are complemented by the high degree of security provided by the usage of LT codes (LT are exposed in Chapter 3.2) as detailed in the following.

The use of LT codes provides two levels of confidentiality. A first level of confidentiality is guaranteed by the fact that every storage node hosts a limited amount of coded fragments from which it is very unlikely to reconstruct the sector content, even assuming that all the private information stored in the corresponding proxy has been hijacked. In the following we refer to this kind of attack as *single storage node attack*. The second form of protection that we analyze is related to the confidentiality of the coding seed z_i that prevent a malicious node, able to collect at least K coded fragments, to gain any knowledge of the sector content. We call this attempt to violate the confidentiality of the data as *sector read attack*. In the next sections we will go through each attack separately.

4.1.1 Single storage node attack

In this case the attacker is a modified storage node that attempts decoding the fragments of a sector she holds for local storage. We can show that ENIGMA is very secure even in the unlikely case that the attacker has come into possession of the coding seed z_i of the sector under attack.

Checking the *hash_id*s of the fragments, an attacker is able to collect all the fragments that belong to a certain sector, even if she is not able to know which sector is. To decode the fragments, the attacker should have the bitmaps linked to the fragments. As already seen, these bitmaps are collected (and encoded) in the proxy table: storage nodes have no information about the bitmaps of the fragments they store.

We can suppose that a modified storage node should be able to obtain such an information. However, even if she knows the bitmaps of the stored fragments, the attacker still can not decode the fragments to obtain the sector. In fact, the number of fragments of the same sector (i.e. labeled with

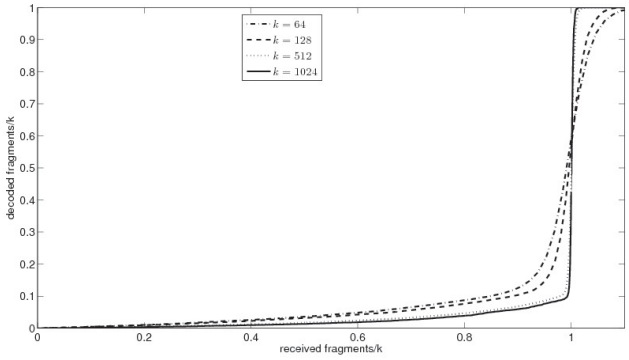

Figure 4.1: Percentage of decoded fragments vs. percentage of owned fragments.

the same *hash_id*) stored in a single storage node are not sufficient to perform a complete decoding: at least K fragment are needed to decode, but each storage node owns far less than K fragments of the same sector.

However, it is possible to attempt to decode at least the fragments owned by a storage node: the process of decoding fountains codes with an insufficient number of fragments is called *partial decoding* [59]. In a partial decoding process, the decoder attempts to decode the maximum quantity of information using the limited number of owned fragments. Gaussian Elimination should be used to perform partial decoding. Figure 4.1 shows the results of the partial decoding of LT codes for k in the range (64,1024); the percentage of decoded fragments is reported as a function of the percentage of coded fragments owned by the node. Both the percentages are calculated with respect to k.

In Figure 4.1 we can see that if a storage node owns a number of fragments far smaller than k, she is not able to decode a large number of fragments, even if she knows their linked bitmaps.

4.1.2 Sector read attack

In the sector read attack we assume that a malicious node has gained sufficient knowledge of the distributed storage system to retrieve a set of K coded fragments of the sector s_i. As already pointed out, such fragments allow the owner of s_i to run the LT decoder and reconstruct the sector content. This is possible because the owner has access to the seed z_i that corresponds to the knowledge of the K bitmaps $b_{i,l}$, $l = 1, \ldots, K$. On the contrary, the attacker has no chance to decode the sector in absence of the bitmap. Indeed, the

attacker could perform a brute force search among all the possible combinations of the bitmaps. For each combination LT decoding can be attempted but the attacker has no means to recognize if the obtained sector is the correct one making the attack impracticable (the attacker does not have context information, the fragments are arbitrary sequence of bits).

This brute force attack turns out to be conceivable only if the CRC of the sector, retained by the proxy, has leaked. In this case the attacker can detect the correct sector by means of the CRC. She could try all the possible combinations and for each one calculate the corresponding CRC, until it matches the correct CRC. However, the large number of required decoding attempts makes the attack unfeasible. Indeed an exhaustive search amounts to test all the possible configurations of K bitmaps, each composed of k bits strings. In the most general case the number of trials turns out to be K^{2^k}. Since we can assume that the attacker knows the degree distribution $P(d = k) = \mu(k)$ used for LT coding a smarter search can be designed. Instead of testing all the possible bitmaps the attackers would limit the search to the most likely outcomes of the bitmap degree. The sequence constituted by K outcomes of the degree can be seen as independent and identically distributed random variables $d_1, d_2, \ldots, d_K$. According to the *asymptotic equipartition property* [28] we can define the typical set of degree outcomes. It is well known that the typical set has probability close to 1, all elements of the typical set are nearly equiprobable, and the number of elements in the typical set is nearly $2^{KH(\mu)}$, where $H(\mu)$ is the entropy associate to the RSD. From the point of view of the attacker the typical set represents the set of the most likely degree sequences to be tested in order break the code. Since the RSD is a peaked distribution where only a limited set of the possible k degrees have non negligible probability, $H(\mu)$ is typically small. As an example, setting the parameters of the RSD $(c, \delta) = (0.01, 0.001)$ [49] one obtains $H(\mu) \approx 3$. It turns out that the typical degree sequences are approximately 2^{3K}. For each degree sequence there are possible combinations of the K bitmaps to be tested. Taking into account the properties of the binomial and that the most likely degree yielded by the RSD is $d = 2$ we can use the following lower bound on the number of combinations

$$\prod_{j=1}^{K} \binom{k}{d_j} \gg \prod_{j=1}^{K} \binom{k}{2} = \left(\frac{k(k-1)}{2} \right)^K$$

We can thus conclude that the number of decoding attempts that the attacker shall perform to break the code is larger than

$$2^{3K} \left(\frac{k(k-1)}{2} \right)^K \tag{4.1}$$

with $K = k + \epsilon$. Clearly, the exponential complexity of the search makes it practically unfeasible. As an example in Figure 4.2 the lower bound of Equation (4.1) is reported as a function of k in the range $(64, 2048)$. It is worth pointing out that for $k = 64$ the number of trial is lower bounded by 10^{300}. This implies that the number of decoding attempts, even for lower values of k, is too high to exploit this type of attack. Moreover, ENIGMA uses values of $k \geq 100$, that are even higher than the aforementioned lower bound.

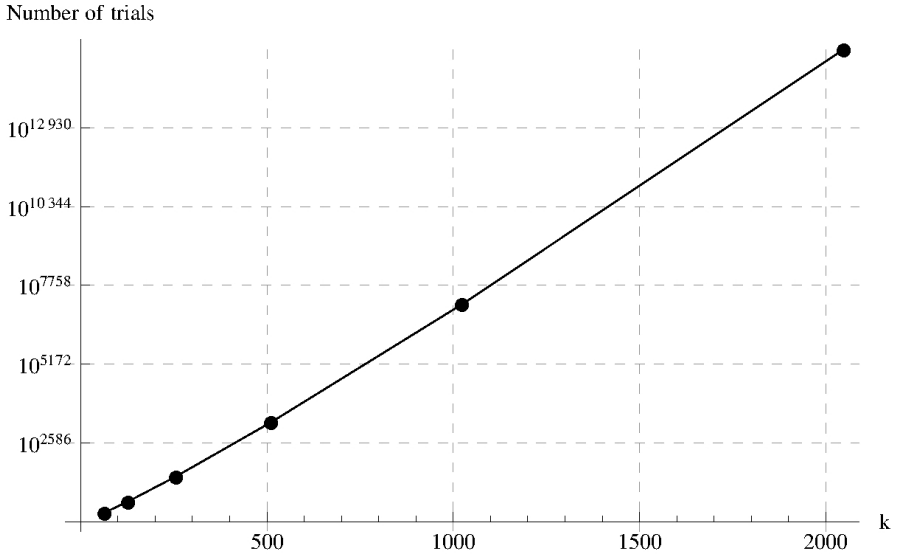

Figure 4.2: Lower bound on the trials required to break the LT code as a function of k for RSD with $(c, \delta) = (0.01, 0.001)$.

4.2 Availability Evaluation

In this section we evaluate the availability level that ENIGMA provides for its virtual disks by first devising an analytical model of availability, and then by using it to quantitatively compute the availability for specific values of n and k. We assume that each sector is split into k fragments, that are subsequently encoded as n fragments, that are stored on n independent nodes. Consequently, when a proxy wants to recover a sector, it asks all the n storage nodes to forward the fragments of that sectors. We also assume that all the storage nodes are characterized by the same reliability p (i.e., the probability of successfully provide a sector fragment to the requesting proxy).

	$k = 64$			$k = 128$		
p	P_k			P_k		
	$n = 83$	$n = 96$	$n = 128$	$n = 166$	$n = 192$	$n = 256$
0.70	0.015	0.302	0.932	0.004	0.500	0.993
0.75	0.091	0.570	0.964	0.085	0.845	0.998
0.80	0.294	0.760	0.981	0.462	0.945	0.999
0.85	0.558	0.860	0.989	0.827	0.977	0.999
0.90	0.741	0.915	0.995	0.931	0.990	1.000
0.95	0.829	0.941	0.996	0.967	0.995	1.000
	$k = 256$			$k = 512$		
p	P_k			P_k		
	$n = 333$	$n = 384$	$n = 512$	$n = 666$	$n = 768$	$n = 1026$
0.70	0.009	0.923	1.000	0.001	0.990	1.000
0.75	0.325	0.996	1.000	0.357	1.000	1.000
0.80	0.919	0.999	1.000	0.989	1.000	1.000
0.85	0.993	1.000	1.000	1.000	1.000	1.000
0.90	0.998	1.000	1.000	1.000	1.000	1.000
0.95	1.000	1.000	1.000	1.000	1.000	1.000

Table 4.1: P_k as a function of p for several values of k and n.

Under these assumptions, the probability $p(r)$ for the proxy to receive r encoded fragments is given by

$$p(r) = \binom{n}{r} p^r (1-p)^{n-r}.$$

As already seen in Chapters 3.2 and 3, the proxy is able to decode the fragments (i.e. to recover the sector) only if $r \geq k$. In particular, if the proxy receives $r \geq k$ fragments it has a probability $\sigma(r)$ to decode the fragments. This probability depends on the rateless code and the decoding algorithm used: in [57] a theoretical study of $\sigma(r)$ for LT codes and Gaussian Elimination-like decoding algorithm is presented. Finally, using the law of total probability, it is possible to calculate the probability that the proxy decodes the requested sector, i.e. the availability, as $P_k = \sum_{i=k}^{n} p(i) \cdot \sigma(i)$.

In Tab. 4.1 P_k is evaluated as a function of the storage nodes reliability p for several values of n and for some values of the code block length $k = 64, 128, 256, 512$. As expected, it can be noted that the availability increases if more coded fragments are stored in system, i.e. if one increases the storage overhead. In Tab. 4.1 the values of n corresponding to a storage overhead of 30%, 50% and 100% respectively have been selected. Tab. 4.1 also shows the

dependency of P_k on the value of k. It can be observed that, given a certain value of p, the required level of availability can be guaranteed increasing k, i.e. cutting each sector into more fragments. This amounts to using an LT code with a larger block size and therefore closer to the channel capacity bound.

The above considerations allow one to derive the basic rules for the fulfillment of a certain service level agreement that, in the most general case, can be varied on a per sector basis. This great flexibility is empowered by the usage of rateless codes where the the sector fragmentation k and the number of coded fragments stored in the system n can be changed adaptively, e.g. in response to the migration of a virtual machine using the cloud disk.

4.3 Performance Evaluation

In this section we provide an evaluation of access time performance of ENIGMA. In the following section we will introduce the simulation techniques that we adopted in order to evaluate the performance of ENIGMA, namely throughput and sector retrieval time. Firstly we suppose that the nodes never fail, then, in section 4.3.4, we will study configurations in which nodes fail with a given probability.

4.3.1 Simulation

As stated in the introduction, we studied the system with simulation techniques. We developed a discrete-event simulator in C++ trying to capture the essential characteristics that correctly describe the system. In order to study the behavior of the system we have to correctly represent the coding process, the network (including the end-to-end latency between the nodes of the infrastructure as well as the end-to-end capacity of the channels in terms of bandwidth) and the behavior of the input process (i.e. the disk workload).

In the remaining of this section we will describe the key features of the simulation design, outlined above.

Sector encoding and decoding simulation

LT codes have an overhead factor ϵ that is related to the number of fragments k. In section 3.2 we discussed that this parameter depends on the fragments (the k chosen among the total number of fragments n) used for the decoding process and it could vary slightly every time the decoding takes place. In order to avoid the development of the decoding process directly inside the

simulator (that could lead to slower simulation time) we decided to draw ϵ from a distribution. Using the technique in [24] we performed several coding and decoding rounds and we collected the values of ϵ; from the data-set collected we extracted the empirical distribution of ϵ for several values of k. Each time we simulate the decoding operation of a given sector, we sample the empirical distribution by extracting a random number between 0 and 1 with uniform probability. In this way we generate the corresponding random variate. The number obtained is the overhead ϵ associated to that particular decoding process.

Latency and Bandwidth distribution

Our system is designed to be deployed on wide area networks and, as already anticipated, we need a method for simulating the key features of a network. We choose to describe the network used by the entity composing the system by means of two characteristics: the end to end latency between each couple of components and the bandwidth.

To get an accurate measurement of the latency we relied on the Meridian distribution [69]. Meridian is a project of the Cornell University that has the goal of building a "peer-to-peer overlay network for performing location-aware node and path selection in large-scale distributed systems". One of the project's feature is the *Meridian data set*, that is a matrix of latencies measured between 2500×2500 nodes. For each couple of nodes a measurement of the nodes end-to-end latency was performed during a period of time of a week. Each value reflects the median of the latency of the collected values. The matrix obtained reflects the average latency of a typical wide area network that, in this case, connects the storage nodes, the cluster heads and the proxy.

In order to give an accurate measure of the system performance we characterized the bandwidth of each node with an estimation that is obtained by samples taken from [44]. In the article the authors conducted an extensive study of the bandwidth measured between PlanetLab [10] nodes. Planet Lab is a globally distributed network with hundreds of nodes spanning over 25 countries and it gives an accurate estimation of bandwidth characteristics of network paths. In the article an end-to-end capacity distribution is provided with an average bandwidth between hosts of nearly $64MBps$.

Disk Workload

In this chapter's introduction we mentioned that we use real word disk traces for the simulations. We motivate this choice by noting that synthetic work-

loads, whereas more flexible, have the inconvenient that they must be extracted from real-world traces, synthesized and then generalized. The drawback is that it is difficult to prove that the most accurate model captures all the important properties of a given set of traces. For these reasons we opted for trace-driven simulation with workload traces chosen accordingly to their characteristics described in [45] and available at [11]. We choose four different workload traces, namely: CFS, RAD-BE, WBS and DAP-DS. The traces' statistical measure are described in detail in the article and have different characteristics:

- CFS: captured on a server that stores metadata for MSN protocol. The trace has a disk access pattern characterized by accesses to multiple files with a high I/O rate (I/O operations per second).
- WBS: captured on a production system that build a complete version of the Windows Server operating system. This trace has a moderate I/O rate, but it is longer in time and transfers more data.
- RAD-BE: captured on a Radius authentication server. It accesses few files with relative high frequency and it has a varying I/O rate.
- DAP-DS: captured at a data server used for caching files for the front-end. It has a low I/O rate and it requests most of the time the same file.

Simulation design details

The first performance metric we wanted to test was average latency of sector retrieval. We modeled this simple setting by creating an entity called storage node and we associated to each storage node a latency chosen uniformly at random from the Meridian data set. The proxy and the cluster heads were part of the infrastructure and they had a latency drawn from Meridian as well. Each fragment belonging to a sector was assigned to a specific storage node with the constraints that a storage node could not store fragments of the same sector. In this way we modeled the request and the retrieval of sectors. This simple simulation environment did not take into account the end-to-end bandwidth of the storage nodes and the proxy, but was accurate to determine the average latency of sectors retrieval.

With these settings, the high number of events generated and the memory requirements for all the data structures, resulted in a slow simulation. For this reason we chose another method for generating sectors retrieval time, so we used order statistics [40]. Given a sample of n variates $X_1, ..., X_N$,

reorder them so that $Y_1 < Y_2 < ... < Y_N$. Then Y_i is called the ith order statistic. If we set X_i as the time for retrieve fragment i of a sector, the Y_k order statistic is the time of retrieval of the whole sector. The sector retrieval time is determined by the time the kth fragments arrives at the proxy.

We then sampled the Meridian data set and the bandwidth distribution to retrieve an empirical continuous distribution that we used to construct the kth order statistic. We used the obtained order statistic to draw a random variate that represent the sector retrieval time, without the need of simulating the retrieval of all the fragments.

In all the simulations we computed the average sector retrieval time with 95% confidence intervals and 2.5% relative error.

In the remaining of this section we compare the performance evaluation in [73] with that of our new simulator. As pointed out in [73] we compared the time taken by a client to retrieve a sector when using, respectively, ENIGMA and a virtual disk located on a single storage server (the *baseline system*). Although very simple, the baseline system can be considered representative of those (very common) scenarios in which the virtual disk must be accessed by the corresponding VM through a production network (e.g., the Internet).

In our experiments, we consider a virtual disk consisting of about 30 millions of 40 KB sectors. We considered an ENIGMA configuration consisting of 2000 perfectly reliable storage nodes, organized into 20 clusters comprising 100 randomly-chosen nodes each; the average latency L of the ENIGMA configuration was $L = 32$ msec. For each sector, we set $k = 100$, and we performed several experiments in which we increased n from k to $3k$ with step 0.1. The resulting fragments were randomly distributed, with uniform probability, over the 2000 storage nodes. The workload used in our experiments is CFS described in section 4.3.1.

The results obtained for the scenarios involving ENIGMA have been compared with those obtained by the baseline system for different values of its *service time* S (the time taken to serve a single sector request). In particular, we performed experiments for $S \in L, L/2, L/4, L/8$. The results of our experiments are reported in Figure 4.3.

As can be observed from this figure, the larger n the smaller the sector retrieval time. This is not unexpected, as a sector is considered to be retrieved when $k + \epsilon$ fragments out of n are received, and the larger n, the larger the probability that the required fragments are shipped by faster nodes.

Figure 4.4 represent the results of the new simulator with the order statistic method. We can observe that the results are similar to that of figure 4.3. The difference in the two figures is due to the delay introduced by the nodes-to-proxy end-to-end bandwidth and the channel capacity of the proxy, as well as a statistical error introduced by the approximation.

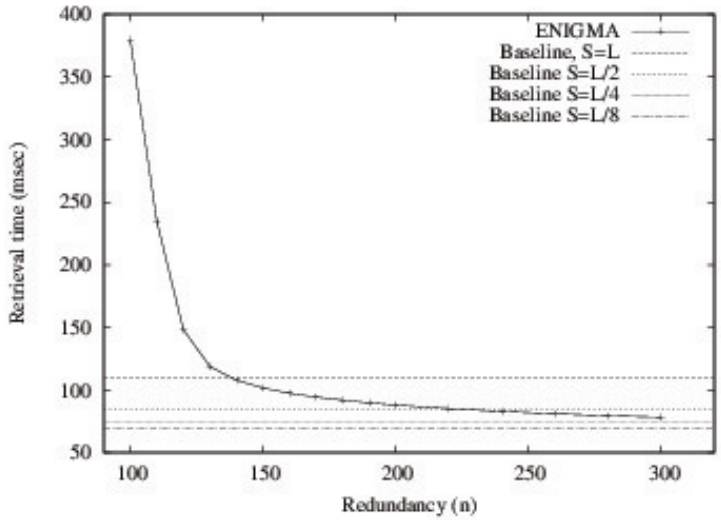

Figure 4.3: Comparison of ENIGMA performance w.r.t. the baseline system.

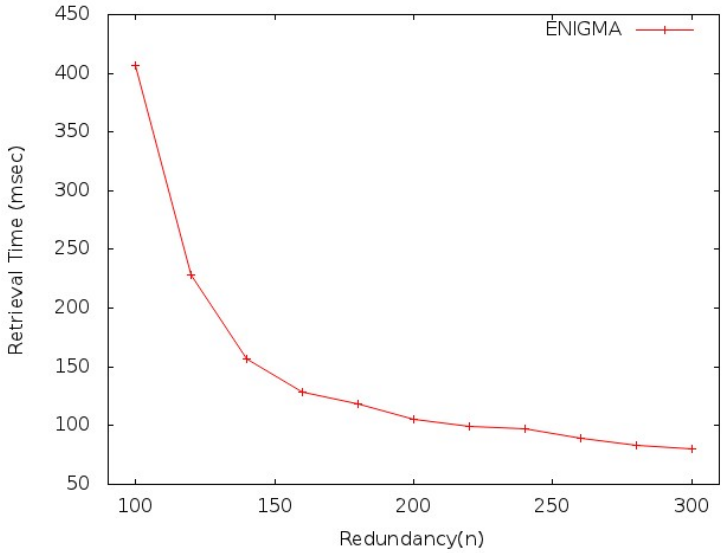

Figure 4.4: Simulation of ENIGMA with order statistic

4.3.2 Sector Access Time Evaluation

In this section we evaluate the disk access performance of read and write operations. Figure 4.5 has the same settings of figure 4.4. As we can see the read access time decreases when we increase the number of fragments per sectors. This is expected because increasing the number of fragments per sector, increases the probability that more fragments will be placed on faster

storage nodes. Since the k fragments needed to recontruct the sector can be on any of the n total fragments, the faster nodes that respond sooner will decrease the sector access time.

On the other hand the write access time (that is the time needed by a write operation to complete) steadily increases. This is expected because the write operation is conservative, meaning that a single write has to wait until all the n fragments are placed on the storage nodes. A less conservative approach is to wait until only a portion of the fragments is stored. For example we could decide to wait until only k fragments plus an arbitrarily chosen epsilon are stored. The other fragments are stored in an asynchronous way, and the write is considered completed before the total n fragments are stored.

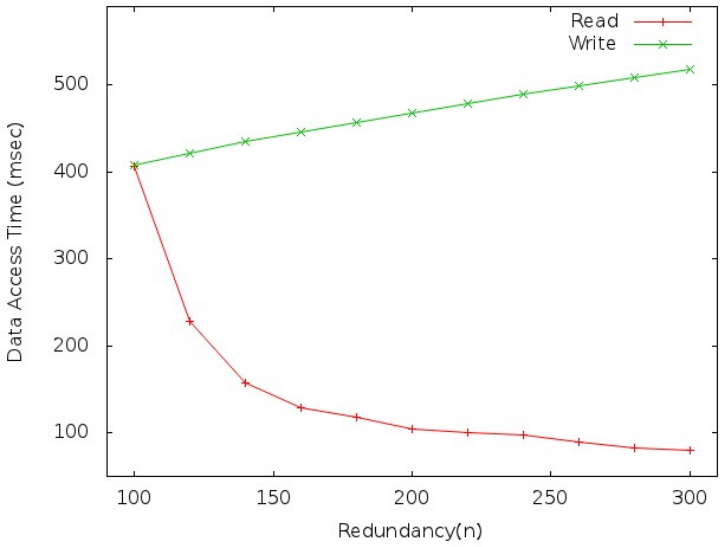

Figure 4.5: Average data access time varying redundancy

4.3.3 Throughput Evaluation

The data transfer rate (throughput) of disk drive (both for read/write) is usually measured by writing a large file to the disk and successively read it.

In the contest of ENIGMA we are interested in the maximum throughput that the infrastructure of storage nodes can provide. For this reason we created two synthetic workloads that request a single file of size $o = 10GByte$ at once (we suppose that the file is allocated on contiguous sectors). In contrast from regular hard disk, whose throughput is limited by the rotational speed and density of the disk drive, ENIGMA throughput is provided by the

network capacity of the infrastructure and by the bandwidth of the proxy. The network capacity of the infrastructure is given by the aggregate network bandwidth of the nodes that store the disk. We measured the throughput of the disk, simulating a read and a write request for the synthetic trace of size o, varying the bandwidth of the proxy, for values ranging from $100MBps$ to $15GBps$. In the next section we will discuss the results for the read and for the write throughput.

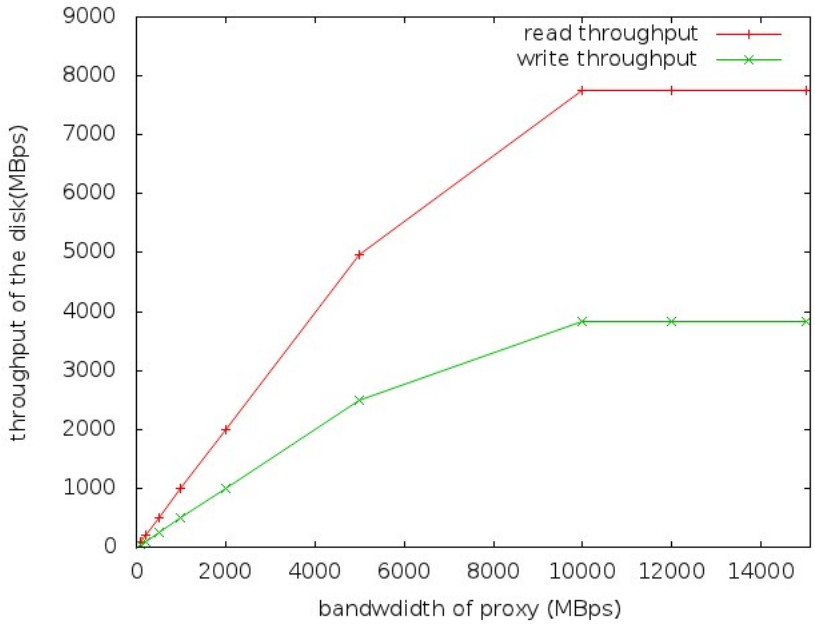

Figure 4.6: Throughput of the disk

Read

As can be seen in picture 4.6, the bandwidth of the proxy is the bottleneck that limits the read throughput of the disk. For bandwidth values up to approximately $8GBps$ the aggregate bandwidth of the infrastructure is able to saturate the proxy channel capacity. Clearly there is a limit in the aggregate network capacity that the infrastructure can provide. This limit arises for values of bandwidth greater than $8GBps$ and is around $7.762GBps$. In the remaining of this section we will provide an explanation for such limit.

A single request is dispatched in parallel by k out of the n nodes that store the fragments of the sector. Two concurrent requests s_1 and s_2 are served in parallel if n_1 and n_2 are two disjoint subsets of nodes composing the infrastructure. But this is not always the case, since the n fragments

of a sector are spread uniformly at random over the infrastructure and thus the two subsets are not disjoint with a certain probability. The real measure of how they overlap is not simply captured by the simulation settings (as described in 4.3.1), specifically by the order statistic method used.

Given this observation we can still determine a lower limit for the bandwidth provided by the infrastructure and show that, even in the worst case, it is sufficient to saturate the bandwidth of the proxy for reasonable values. In the worst case all the sectors are spread on the same n nodes of the infrastructure and thus the amount of capacity is that of such nodes in parallel. Even in this scenario, the aggregate bandwidth of the nodes is able to provide a throughput of $7.762 GBps$ as can be seen in figure 4.6.

Write

In picture 4.6 we can see that the throughput of the write operation is always half of the read throughput. This could be explained by noting the particular settings used for this set of simulations. In the simulation we used $k = 128$ and $n = 256$. Since ENIGMA has to wait that all the n fragments are stored in order to consider the operation completed, the available throughput is halved ($3.874 GBps$).

4.3.4 Fault tolerance Evaluation

In this section we show the results of a series of simulation with a configuration in which the nodes can fail. A failure means that, upon receiving a request for a fragment, the storage nodes does not reply to that request. Each node has a uniform at random probability p of responding to a request (a perfect reliable node has a probability of response $p = 1$). We performed the simulations varying p ($p = 0.5$, $p = 0.7$ and $p = 0.9$) as well as the redundancy of each sector, with fixed $k = 100$ and varying $n = 100, \ldots, 200$ with an incremental step of 10 fragments. Similar to real disk drives, we set a fixed timeout for the read request. After a sector is requested we wait until either the sector is retrieved or the timeout occurs. In the latter case we request again the fragments of the sector that have not yet been retrieved (due to a storage node failure) and we wait for another timeout. If the timeout occurs again we have a failure and the infrastructure is not able to deliver the sector (in practice we give a second chance to a request after the first time occurs). The results follow what was anticipated theoretically in section 4.2. If redundancy is proportional to the failure rate ENIGMA is able to deliver the sector correctly.

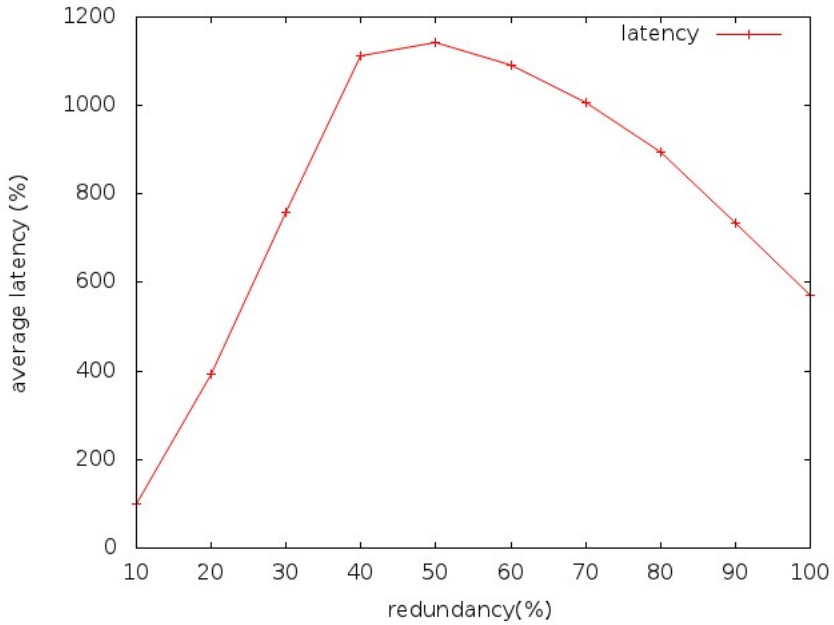

Figure 4.7: average latency when varying redundancy with probability of response $p = 0.5$

Unreliable nodes ($p = 0.5$)

In Figure 4.7 we can see how latency varies with varying redundancy and a set of nodes that have a probability of response $p = 0.5$. When redundancy is low, the response time is low, because we expect an high failure rate of the nodes and so the timeout occurs for almost all the sectors requested. Only a small percentage of sectors (Figure 4.8) is retrieved and corresponds to the sectors with fragments placed on faster nodes. As redundancy increases (form 10% to 40% of fragments), the response time increases, because more sectors are correctly delivered. In this case, the redundancy can handle the failure of the nodes, but at the cost of a higher average latency, that is determined by the high number of timeouts that still occurs. When redundancy increases further (from 50% to 100% of fragments), the failure rate of requests drops to zero and, conversely, the requests successfully delivered become 100% of the total requests. This is somewhat expected because, intuitively, a redundancy of 50% is able to tolerate a failure rate of 50%. It is worth noting that, even if all the sectors are delivered (100% of responses), we still observe an high number of timeouts (50% of the responses), because the fragments in excess are used to cope with node failures and they do not decrease sectors access time.

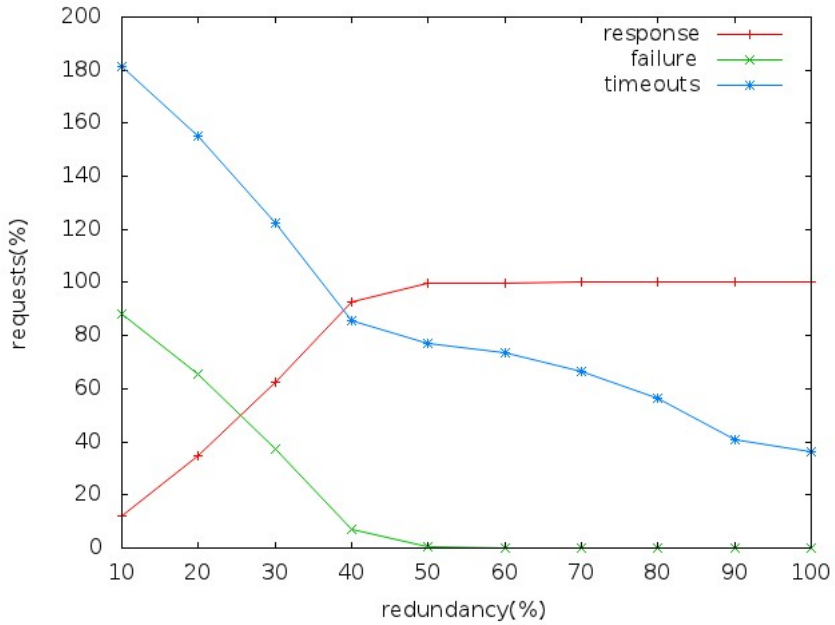

Figure 4.8: percentage of responses timeouts and failures with probability of response $p = 0.5$

Moderately reliable nodes ($p = 0.7$)

In this scenario, we evaluate the latency of sector retrieval when the response probability of the set of storage nodes is moderately high $p = 0.7$. As we can see in Figure 4.9 and 4.10 an increment of redundancy of 30% can cope with failures and succeed in delivering the 100% of sectors requested. From this point on, the average sector retrieval time constantly decreases when redundancy increases. The decrement of sector retrieval time is not as fast as we expect though, because node failures still introduce errors that cause timeouts. The percentage of timeouts drops to 0 when redundancy is greater than 70% of increment (Figure 4.10).

Highly reliable nodes ($p = 0.9$)

Figure 4.11 and 4.12 correspond to a scenario in which nodes reply to requests with high probability. Such a scenario could correspond, for example, to a reliable set of nodes located on a controlled datacenter. In this case, even a low redundancy (10%) is able to provide enough reliability and, as we can see, in all cases we have a 100% of successful replies. Differently from the previous cases, an increase of redundancy brings immediately an increase of the average sector retrieval time, because the fragments are used to decrease

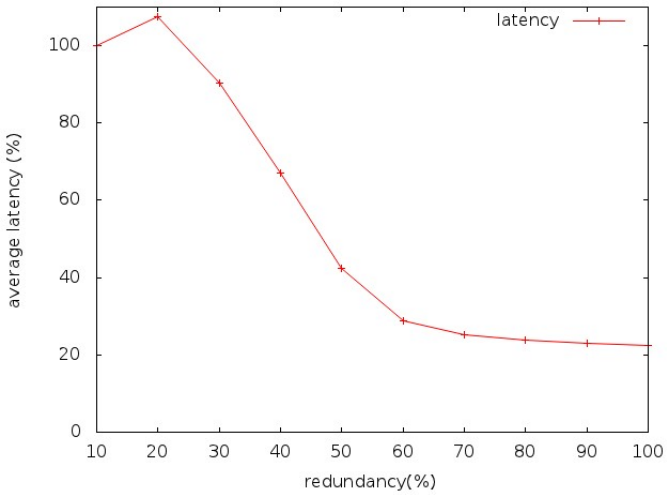

Figure 4.9: average latency when varying redundancy with probability of response $p = 0.7$

access time. In particular we observe the absence of timeouts that would have slowed down the sector retrieval time.

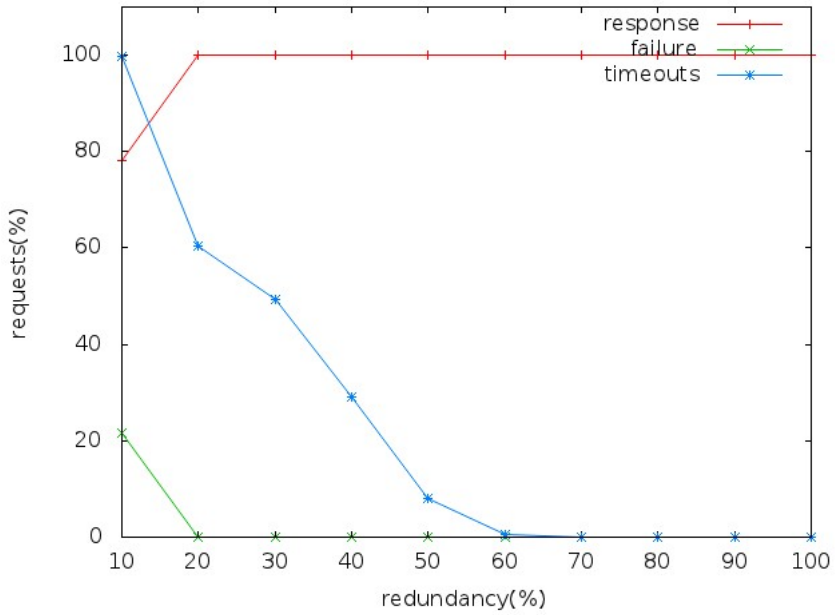

Figure 4.10: percentage of responses timeouts and failures with probability of response $p = 0.7$

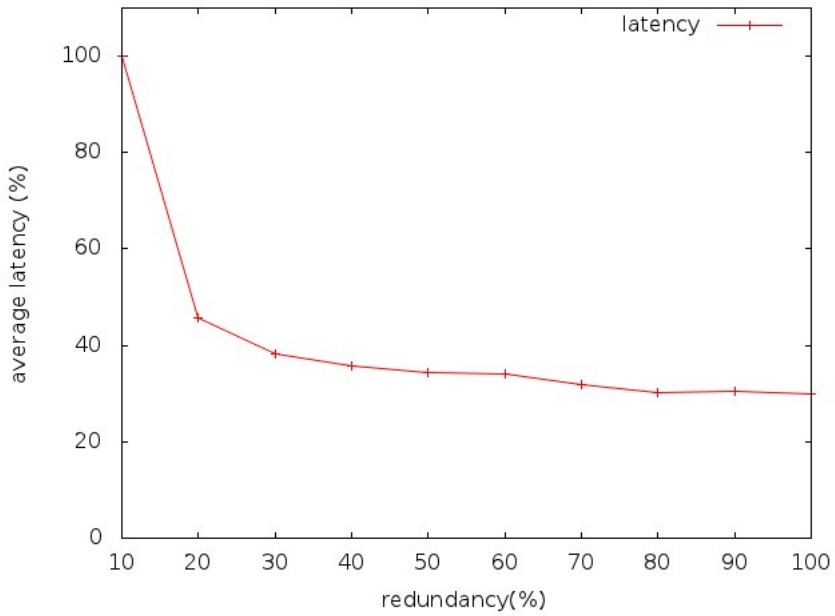

Figure 4.11: average latency when varying redundancy with probability of response $p = 0.9$

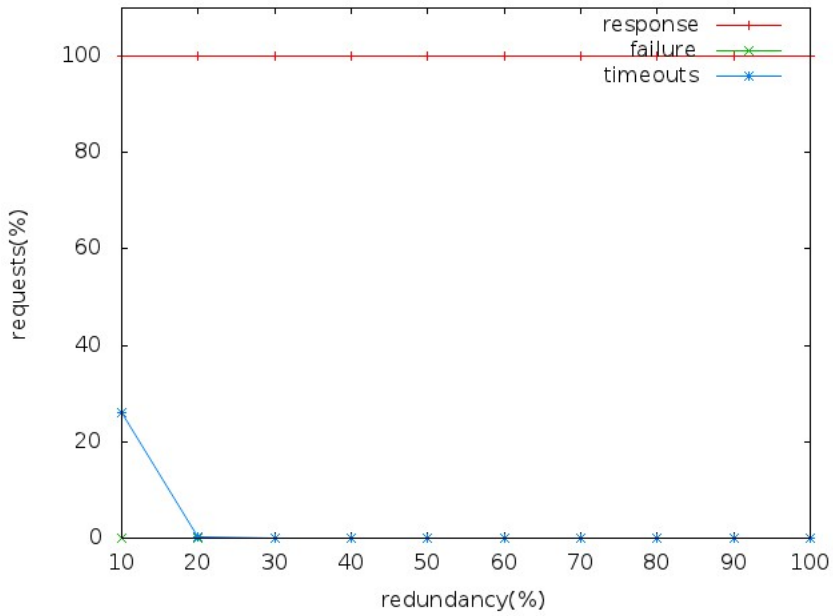

Figure 4.12: percentage of responses timeouts and failures with probability of response $p = 0.9$

Chapter 5

Caching architecture and policies in ENIGMA

In this chapter we describe the caching mechanisms for the ENIGMA system. Caching techniques are a standard mechanism used for decreasing data access time and increasing throughput. They have been widely adopted in various contexts, such as disk drives, processors, databases, web servers, file systems, operating systems and distributed systems. A cache is a memory that can store a limited amount of data, but guarantees better access time than the primary storage. Storing the most accessed data items in cache allows performance benefits, but the limited amount of space raises the problem of choosing which elements hold and which eliminate.

5.1 Caching in Enigma

In this section we provide an exposition of caching mechanisms and policies that will be used in ENIGMA as a mean to improve performance.

Two aspects characterize a caching system: its size and the replacement algorithm (that chooses which element keep in cache and which erase). A large body of literature has extensively studied replacement algorithms, but the distinctive features of our system do not allow to determine a priori if the results achieved by those algorithms for traditional systems are valid applied to ENIGMA. ENIGMA, unlike real hard disk drives, has a different notion of sector contiguity; in traditional disks contiguous sectors are allocated in the same track and in the same cylinder, and operating systems tries to exploit this peculiarity by storing files in contiguous sectors, in order to reduce seek time and the repositioning of mechanical parts of the disk drive. This assumption, on which file systems are based, has lead to the development of

caching and prefetching strategies that work well on real systems. ENIGMA on the contrary does not have a notion of proximity based on mechanical parts, hence apparently close sectors may in fact be stored on different storage nodes. In this context the miss penalty (i.e. the cost of retrieving a sector from the storage nodes, in case it is not in cache) varies significantly and is related to the placement of the fragments over the infrastructure. Traditional caching policies are applied in a context (disk drives) where the response time has a predictable pattern, that depends only on the geometry of the disk and the stream of requests. In a network the response time varies dynamically (and with higher variance) not only due to the workload, but also for external factors (such as for example network congestion, load on the servers and nodes failure). For these reasons we will study each policy applied in the context of ENIGMA. In the next sections we will describe the architecture and the policies of Enigma caching system.

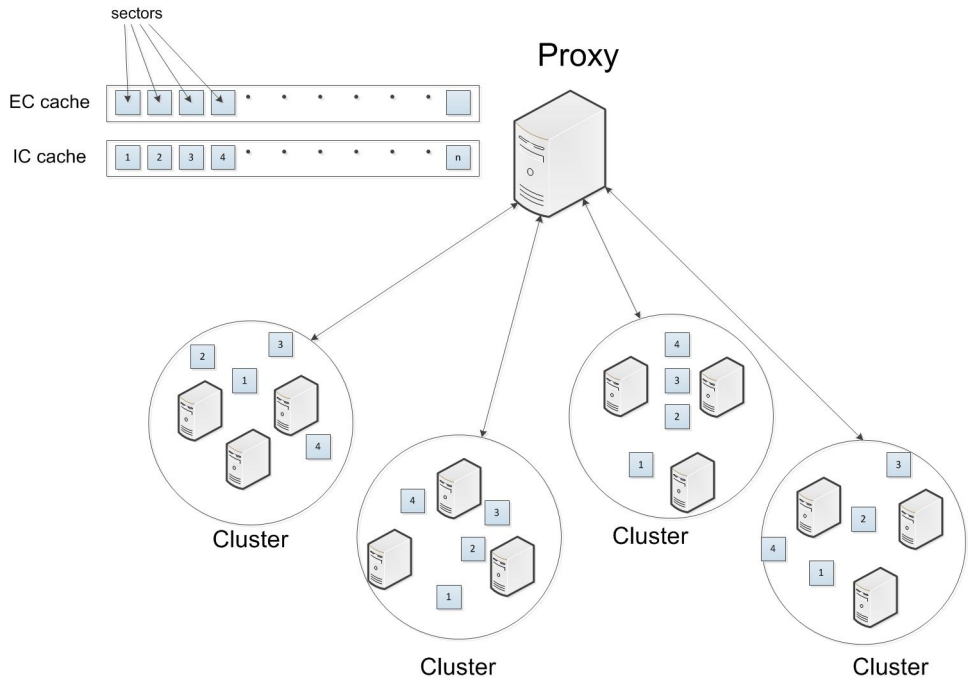

Figure 5.1: Enigma caching architecture

5.1.1 Caching architecture

Enigma caching architecture is structured in two levels (Figure 5.1). A first level cache, called Explicit Cache (EC for short), is a classical software cache and is located on the proxy; it has the task of storing decoded sectors. When

a client issues a request, the proxy first looks into EC cache and, if the sector is present, delivers it to the client (in that case we have a cache hit). If the sector is not present in EC, the proxy retrieves it from the storage nodes and subsequently stores it into the cache. The amount of storage assigned to this cache is limited by the storage capacity of the proxy. If the cache is full, a "victim" is chosen to be evicted and leave room for the sector just retrieved. The second level cache, called Implicit Cache (IC for short), is not a proper software cache, but it exploits the power of coding techniques in order to decrease sectors' retrieval time. In practice IC cache increases selectively the redundancy of some sectors with the consequence of decreasing retrieval time (increasing redundancy decreases access time, as pointed out in [73]). The sectors with augmented redundancy constitute the IC cache. In this case the sectors are not stored in the proxy, but rather the proxy maintains a list of references to the sectors that form the IC cache. The dimension of the cache is the number of sectors present in the list. In this case the amount of storage dedicated to the cache is that of the storage nodes, that have to store the additional fragments of the "cached" sectors. When the proxy decides to put a sector in IC, the redundancy of that particular sector is increased; conversely, if IC is full (i.e. the number of sector in the list equals the predefined dimension of the cache), a sector is chosen to be expelled and, consequently, its redundancy is decreased to the standard level. The mechanisms for increasing and decreasing sectors redundancy are described in details in section 3.2.

In the remaining of this section we will describe caching policies (i.e. replacement algorithms) and redundancy mechanisms in order to implement the caching system of Enigma. In section 5.2 we will evaluate these policies by simulation.

Two classes of algorithms are used in the context of caching: caching algorithms and prefetching algorithms. Caching tries to capture temporal locality of sectors requests, while prefetching tries to capture spatial locality. Locality of references is a principle that correlates subsequent accesses to storage locations. Temporal locality refers to the concept that a storage location that is referred at some time, will be likely referred again in the near future. Spatial locality, conversely, refers to the likelihood of accessing data items in close storage locations. In the context of ENIGMA we apply traditional caching algorithms to EC while we use prefetching in IC. The rationale behind this choice is that the cost of read ahead is higher if we fetch the sector rather than if we increase its redundancy. That is, fetching a sector to the cache needs an higher time than increasing the sectors' redundancy level. In practice, when a request arrives for sector s_i, the time for retrieving subsequent n sectors has a certain value, say t_f. If we choose to increase the

redundancy of the n sectors instead, the time for completing the operation (considering that is performed by the infrastructure and not by the proxy) is t_r, with $t_r < t_f$ on average (remember that only a portion of the fragments is needed to increase redundancy and not all the fragments as in the case of reconstructing the whole sector). In case prefetching is too much aggressive, for example because the workload shows low spatial locality, the cache is polluted with useless sectors (i.e. sectors that will not be requested in the future). If we fetched the sectors to the cache we would have wasted precious resources and time; if conversely we increase redundancy the disadvantage of useless prefetch is diminished. Prefetching with the mechanism of increasing and decreasing redundancy could be a good trade off between increasing performance and use of storage resources. Moreover if we decide to fetch sectors contiguous to the one requested, the time needed for the operation will be likely greater than the time when the sectors will be referenced.

For these reasons, in order to gain an advantage from the prefetching mechanism, we choose to increase the redundancy of sectors instead of fetching them to the cache. Other algorithms tries to combine caching and prefetching in a single cache. We applied these algorithms to the IC in situations where an EC cache could be not available (for example when the disk is accessed with a tablet pc).

In the next paragraphs we will describe caching and prefetching algorithms considered for ENIGMA.

5.1.2 Explicit Cache algorithms

As already mentioned, several caching algorithms are known in the literature. An overview and a comparison study of such algorithms is given in [72] and [52]. All the algorithms listed below exploit the principle of temporal locality of data, according to which it is likely that a requested sector will be referenced again in a relatively short amount of time. The various algorithms are variations of the classical Least Recently Used (LRU) algorithm, that tries to capture recency, and the Least Frequently Used (LFU) algorithm that tries to capture frequency of accesses. In practice LRU always replace the least recently used item in the cache. The cache is seen as a list of items, so every time a reference is made to an element, this one is placed on the head of the list while the last element (the least recently used) is evicted. Conversely LFU counts references to the elements in the cache and removes the one with the lowest value. Most of caching algorithms are variation of these two, we will evaluate the most recent ones found in the literature. The algorithm that we choose for comparison are: LRU, 2Q [43], LIRS [42], ARC [52] and CAR [21].

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1

Table 5.1: List of requests

In the next paragraphs we will explain the key ideas of each caching algorithm. We will use a simple example to illustrate the inner working of each algorithm. The example is a stream of sector requests, depicted in Table 5.1. Each cell in the list is a request for the sector that is contained in the cell; the sectors in the list are requested from left to right. For each algorithm we will explain how the algorithm behaves in response to this stream of requests.

LRU

LRU maintains a list, managed as a stack, of references to the sectors. The head of the stack represents the most recently used (mru) element of the list. When an item is accessed it is placed on the mru section of the list and when the list is full, the algorithm evicts the sector at the tail of the list, that is the least recently used (lru) sector. Table 5.2 depicts a working example of LRU. The list of requests (table 5.1) is depicted in the top row of table 5.2. In 5.2a through 5.2o it is shown the behavior of a LRU cache of size 3 (the symbol (∗) marks a cache hit). Each column represents the content of the cache at the time the corresponding request arrives. The top cell of each column represents the most recently used element in the cache, whereas the bottom cell represents the least recently used element. Table 5.2a represents the first three steps of the algorithm, each element is placed at the top of the cache and no other action is performed, because the cache is empty and there is room for storing the requested sectors. When the fourth request arrives the cache is full (Table 5.2b) and the algorithm performs the first eviction. The lru element (bottom of the cache) is evicted and sector 2 is placed in the mru position of the list. The fifth requests is a hit (Table 5.2c), because 0 is already an element of the cache, hence it is moved to the mru position of the list. The next request (Table 5.2d), for sector 3, is a miss. The sector is not in cache and thus the lru element of the cache is selected for eviction and sector 3 is moved to the mru position. Subsequent request (Table 5.2e) is a hit again (sector 0 is still in cache). The next four requests (Table 5.2f to Table 5.2i) are all misses, thus the lru element (bottom) of the list is evicted and the requested sector is placed in the mru position (top) of the list. Successively we have two hits, because requested sectors 3 and 2 are present in the cache (Table 5.2j and Table 5.2k). The remaining requests are in order, a hit (Table 5.2l), a miss (Table 5.2m), a hit again (Table 5.2n) and another miss (Table 5.2o).

Cache references:

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
				(*)		(*)					(*)	(*)		(*)		(*)
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0
		7	0	1	2	2	3	0	4	2	2	0	3	3	1	2
(a)			(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)

Table 5.2: LRU example

LRU algorithm is really simple and we will use it for comparison purposes only.

2Q

```
// If there is space, we give it to X.
// If there is no space, we free a page slot to
// make room for page X.
reclaimfor(page X)
begin
    if there are free page slots then
        put X into a free page slot
    else if(|A1in| >Kin)
        page out the tail of A1in, call it Y
        add identifier of Y to the head of A1out
        if(|A1out| >Kout)
            remove identifier of Z from
            the tail of A1out
        end if
        put X into the reclaimed page slot
    else
        page out the tail of Am, call it Y
        // do not put it on A1out; it hasn't been
        // accessed for a while
        put X into the reclaimed page slot
    end if
end
```

```
On accessing a page X :
begin
    if X is in Am then
        move X to the head of Am
    else if (X is in A1out) then
        reclaimfor(X)
        add X to the head of Am
    else if (X is in A1in)      // do nothing
    else      // X is in no queue
        reclaimfor(X)
        add X to the head of A1in
    end if
end
```

Figure 5.2: 2Q algorithm

LRU works well because tends to remove "cold sectors" (i.e. sectors that are not accessed for longer time) to make space for the requested sector. If, however, the requested sector is cold, LRU may remove "warmer sectors" (i.e. sectors that are supposed to be accessed in the near future) from the cache to store the requested (cold) sector, that will remain in the cache for a long period of time. 2Q improves upon LRU, because it uses a main buffer

Cache references:

		7	0	1	2	0	3	0	4	2
								(∗)		
Am(2)						0	0	0	0	2,0
A1in(1)		7	0	1	2	2	3	3	4	4
A1out(2)			7	0,7	1,0	1	2,1	2,1	3,2	3
(a)	(b)		(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)

	3	0	3	2	1	2	0	1
	(∗)	(∗)	(∗)	(∗)		(∗)		(∗)
Am(2)	3,2	3,2	3,2	2,3	2,3	2,3	0,2	0,2
A1in(1)	0	0	0	0	1	1	1	1
A1out(2)	4	4	4	4	0,4	0,4	4	4
(k)	(l)	(m)	(n)	(o)	(p)	(q)	(r)	(s)

Table 5.3: 2Q example

in which only warm sectors are stored and a special buffer to place sector referenced for the first time.

This algorithm uses special queues: A1 and Am. A1 is further divided in two queues A1in and A1out. Am is managed as an LRU list, and is used to store the elements that have a steady frequency of accesses (i.e. they are requested regularly over time). A1in, instead, is used to keep the sectors that are accessed with high frequency, probably due to correlation in the references (a correlated reference is a close in time access to the same sector, probably made by the same process). The elements from A1in not accessed for a longer time are placed in a temporary ghost queue, A1out. A1out maintains only the reference to the sectors, not the sectors themselves. It is used to detect sectors with long-time access rates. The size of the two queues, A1in and A1out, must be chosen carefully, in an off-line fashion. This could be potentially a difficult task and, for this reason, the authors suggest two typical values that performs well in most of the situations. In a typical setting, given a cache of size c, A1in should be 25% of c, leaving the remaining space to Am. A1out instead should be as large as 50% of the cache size c. In Figure 5.2 we can see the detailed algorithm for 2Q.

In Table 5.3 we can see the sample list of sector requests and how 2Q act in response to these requests (the symbol (∗) indicates a cache hit). Table 5.3a represents each queue described in the previous paragraph. Each cell, corresponding to a particular queue, represents the list of sectors present

in the queue at a particular time. It is a comma separated list of sectors, and, as a rule, the head of the queue is the leftmost sector, while the tail of the queue is the rightmost one. When the cache is empty each request is served by the A1in queue and the requested sector is placed in the head position of A1in. When A1in is full (after the first request in this example because the size of A1in is one) the tail of A1in is evicted and an identifier is placed at the head position of A1out. This behavior is depicted in Tables 5.3b to 5.3e, in which all the requests are miss (the requested sectors are not already cached). In Table 5.3f we have another miss, but this time the requested sector is present in the A1out queue, which means that we have a long time reference correlation. Sector 0 is then moved to the head of Am queue. The next request 5.3g is a miss, sector 3 is placed in A1in and an identifier is added to A1out with a reference to the sector in the tail of A1in (sector 2 in this case, that is removed to make room for the requested sector). Next reference, depicted in Table 5.3h, is a hit, because sector 0 is in Am queue. The sector is moved at the head of Am (in this case nothing happens because it is the only sector present). For the request of sector 4 (Table 5.3i) the algorithm places the sector in A1in and an identifier is added to A1out (a reference for the sector in tail position of A1n). In Table 5.3j the requested sector is reference in A1out (sector 2), and it is promoted to the Am queue. The subsequent four requests (Table 5.3l through 5.3o) are all hits to the Am queue, which means that the referenced sector is moved to the head of the queue. Successive request is a miss (Table 5.3p) and sector 1 is placed in the A1in queue (sector 0 is evicted from the cache and a reference is placed in A1out), then we have a hit for sector 2 (Table 5.3q), that leaves the cache unchanged. The next request is a miss, sector 0 is not in cache, but it is referenced in A1out. For this reason the sector is moved in the mru position of Am (Table 5.3r). Finally, the last request is for sector 1 (Table 5.3s), that is present in A1in (hit).

LIRS

2Q is an attempt to cope with LRU inability to deal with access patterns with weak locality, but a the cost of increasing the algorithm complexity and with the addition of parameters that have to be carefully tuned. LIRS tries to address the limits of LRU and 2Q, by using recency to evaluate Inter-Reference-Recency (IRR) for making a replacement decision. The IRR is the recording information associated to each sector and represents the number of other sectors accessed between two consecutive references to the sector; conversely, recency is the number of requested sectors between the last reference and the current time. LIRS uses this information in order to

Cache references:

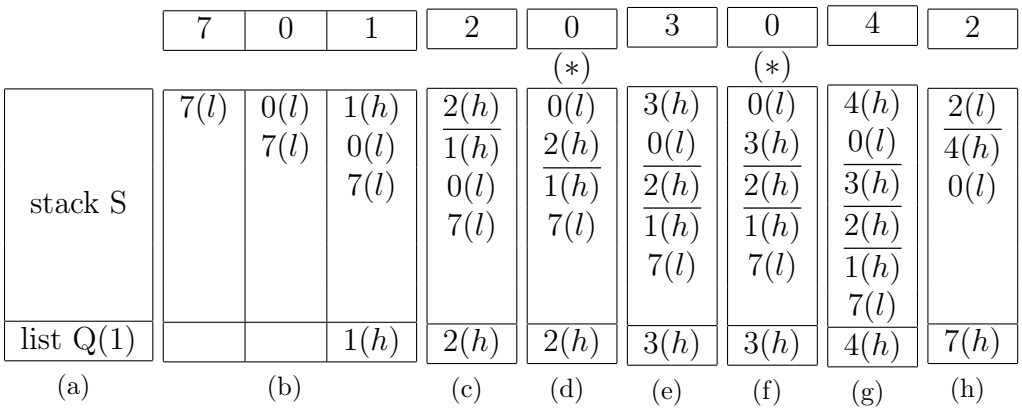

Table 5.4: LIRS example

decide which sector evict. The algorithm distinguishes between sectors with low IRR (denoted as LIR) and sectors with high IRR (denoted as HIR), the recency is used only to determine the LIR or HIR status of each sector. LIRS maintains a LIR set and a HIR set, and manages to limit the size of the LIR set so that all the sectors belonging to the set fits the cache. The HIR set, conversely, resides on a small portion of the cache and its sectors can be evicted at any recency.

LIR set is completely contained in the cache whereas HIR sectors could be either in cache (resident HIR sector) or not (non resident HIR sectors). LIRS divides the cache in two parts, the largest part (of size L_{lirs}) is used for storing LIR sectors, and the smallest part (of size L_{hirs}) used for storing resident HIR sector ($L_{lirs} + L_{hirs}$ equals cache size). The authors suggest L_{hirs} to be 1% of total cache size. The ability of LIRS is to maintain and dynamically adapt the LIR set and HIR set. This is accomplished by switching the sectors status between LIR and HIR, based on the comparison of the IRR of each sector. If a HIR sector is referenced, it gets a new IRR status that is equal to its recency. If the IRR is smaller than the recency of a LIR sector it means that the next IRR of the LIR sector will be greater than that of HIR sector. This assumption holds because the recency of the LIR sector is part of its incoming IRR and no greater than the IRR. Once LIRS has found that the maximum recency of a sector in the LIR set is greater than the IRR of the HIR sector, it switches the status of the two sectors.

In order to avoid the burden of maintaining the information relative to the IRR and recency of each sector, LIRS algorithm is implemented using a lru stack, namely stack S, that records the recency using the position in the stack as the lru algorithm does. Stack S has varying size and records the LIR or HIR(either resident or non-resident) status of sector that are pushed into it. The resident HIR sectors are further recorded in a list, called listQ, with maximum size L_{hirs}. In stack S the bottom sector is always a LIR sector and, if it is removed (because its status is swapped with a HIR sector), LIRS perform an operation of stack pruning, that removes all the elements in the stack until a LIR sector is found. This operation is performed because HIR sectors above the bottom LIR sector have a recency lower than the other LIR sector in the stack and thus they will never get the chance to switch their status to LIR.

When the cache is empty, to all the referenced sectors is given the LIR status, then, when the cache is full (the number of LIR sectors equals L_{lirs}), the algorithm performs as follows:

- **access to a LIR sector X:** this is a hit, the sector is moved to the mru position of stack S and pruning is performed if needed.

- **access to a HIR resident sector X:** this is a hit, the sector is moved to the mru position of stack S and two cases are possible: (1) the sector is in stack S and thus its status becomes LIR (then the LIR sector in the lru position of stack S becomes a HIR sector). Stack pruning is performed if needed. (2) X is not in stack S, its status is set to HIR and it is placed in list Q.

- **access to a HIR non-resident sector X:** this is a miss, the sector is moved to the mru position of stack S and the HIR sector in mru position in listQ is removed. Two cases are possible: (1) the sector is in stack S and thus its status becomes LIR (then the LIR sector in the lru position of stack S becomes a HIR sector). Stack pruning is performed if needed. (2) X is not in stack S, its status is set to HIR and it is placed in list Q.

The working example of LIRS is depicted in Table 5.4. We will provide a step-by-step description of the algorithm applied to the list of requests of Table 5.1. Each column (Tables 5.4b to 5.4q) corresponds to a step of the algorithm and the requested sector is provided in the corresponding cell atop of each column. In table 5.4a, the "stack S" reading direction is from top to bottom and has varying size; "list Q" is of size one. When the cache is empty, the new sectors are assigned the LIR status (an (l) next to the sector number), until the LIR set is full (2 sectors in this case); at that point, to the new requested sectors is given the HIR status (an (h) next to the sector number). The first three steps in the example (table 5.4b) depict this situation; sector 1 (the last sector requested) is on top of stack S with status HIR (and at the same time is referenced in list Q). The former requested sectors, 0 and 7, are of LIR type and are placed in the bottom position of stack S. The successive request (sector 2) is a miss, 2Q evicts the HIR sector from the cache (sector 1 becomes a non-resident HIR sector in stack S, marked as $\overline{1(h)}$ in table 5.4c) and 2 becomes the new resident HIR sector. The next request, for sector 0, is a hit and the sector is moved to the top position of stack S (table 5.4d). Successive request for sector 3, is a miss and the sector becomes the HIR resident sector (table 5.4e). The request for sector 0 is a hit again and the sector is moved to the top position of the stack (table 5.4f); the successive request for sector 4 is a miss and the sector become the new HIR sector (table 5.4g). The next request is more interesting from the point of view of the algorithm (table 5.4h), because sector 2 is a non resident HIR sector (sector $\overline{2(h)}$ in stack S). Accessing a HIR sector (that is present in the stack) means that its recency is lower than the recency of the LIR sector that resides at the bottom of the stack. The algorithm then switches the status

of the two sectors and move the former LIR sector (now HIR sector) to the tail position of list Q. Then it moves the new LIR sector to the top of the stack and perform a stack pruning in order to ensure that the bottom sector of stack S is a LIR sector. Request for sector 3 is a miss, it becomes the new HIR sector and is placed on top of stack S and in the head position of list Q (table 5.4j). The next three requests are all hits to a LIR sector (tables 5.4k, 5.4l and 5.4m); in all three cases the requested sector is moved to the top of stack S with its status unchanged and stack pruning is performed if needed. Request for sector 1 is a miss and the sector becomes the new HIR sector (table 5.4n). The next request is a hit and sector 2 is moved to the top of stack S (table 5.4o). Successive request (sector 0) is a hit, but this time the sector is the bottom sector of stack S; in this case the algorithm moves the sector on top of stack S and performs a stack pruning, deleting from the stack sectors 1 and 3 (table 5.4p). Finally in the last request the sector has the HIR status (thus we have a hit) but the sector is not present in stack S. For this reason the algorithm does not have any notion of the recency associated to the sector (the sector is treated as if it were accessed for the first time) and no status change is performed (table 5.4q).

ARC

ARC algorithm is an attempt to dynamically adapt the caching policy to the evolving and continuously changing access patterns of the workload. This algorithm uses a learning rule to dynamically balance between recency and frequency patterns in an on line and self-tuning fashion and it tries to avoid the use of user-defined parameters (used for example in 2Q). ARC maintains two lists (managed as lru lists): one that stores sectors accessed only once recently, while the other list stores sectors that have been requested at least twice recently. In practice one list tries to capture recency, while the other tries to capture frequency. The size of the two lists together can reference exactly twice the number of sectors that the cache can store and, at any time, ARC chooses a variable number of sectors taken from the most recently used portion of the two list. The amount of sectors that ARC chooses from each list is a parameter that is adaptively inferred from the recent history of accesses.

In Figure 5.3 we can see the details of the ARC algorithm. ARC maintains 4 lru lists: T1, B1, T2 and B2. The real cache is formed by the sectors contained in lists T1 and T2. The algorithm dynamically decides, based on the observed requests it receives, which item replace and in which list. On a cache miss, ARC dynamically decides whether to replace lru sector in T1 or lru sector in T2, depending on the value of the parameter p. The general

ARC(c)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists T_1, B_1, T_2, and B_2 to empty.

For every $t \geq 1$ and any x_t, one and only one of the following four cases must occur.
Case I: x_t is in T_1 or T_2. A cache hit has occurred in ARC(c) and DBL($2c$).
Move x_t to MRU position in T_2.

Case II: x_t is in B_1. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_1 to the MRU position in T_2 (also fetch x_t to the cache).

Case III: x_t is in B_2. A cache miss (resp. hit) has occurred in ARC(c) (resp. DBL($2c$)).

ADAPTATION: Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

REPLACE(x_t, p). Move x_t from B_2 to the MRU position in T_2 (also fetch x_t to the cache).

Case IV: x_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC(c) and DBL($2c$).

Case A: $L_1 = T_1 \cup B_1$ has exactly c pages.
If ($|T_1| < c$)
Delete LRU page in B_1. REPLACE(x_t, p).
else
Here B_1 is empty. Delete LRU page in T_1 (also remove it from the cache).
endif
Case B: $L_1 = T_1 \cup B_1$ has less than c pages.
If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
Delete LRU page in B_2, if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
REPLACE(x_t, p).
endif
Finally, fetch x_t to the cache and move it to MRU position in T_1.

Subroutine REPLACE(x_t, p)
If (($|T_1|$ is not empty) and (($|T_1|$ exceeds the target p) or (x_t is in B_2 and $|T_1| = p$)))
Delete the LRU page in T_1 (also remove it from the cache), and move it to MRU position in B_1.
else
Delete the LRU page in T_2 (also remove it from the cache), and move it to MRU position in B_2.
endif

Figure 5.3: ARC algorithm

idea of the algorithm is that if the p tends to 0 the algorithm tends to favor T2 over T1; if, conversely, p tends to the maximum size of the cache, ARC tends to favor T1 over T2. The lists B1 and B2 maintain a reference to the sectors (the referenced sectors are not in cache) evicted from respectively T1 and T2, and are used in the adaptation process.

The policy continuously adapt the parameter p on each request. Intuitively, if there is a hit in B1 the size of T1 should be increased; conversely if there is a hit in B2 the size of T2 should be increased. Hence, on a hit in B1 the parameter p (that represent the target size of the list) is increased and, on a hit on B2 is decreased (if c is the total size of the cache $p - c$ is the target size of list T2). The amount of increment (or decrement) of p is the learning rate of the algorithm and depends on the size of lists B1 and

B2. On a hit on B1 the increment is proportional to the relation between the size of B1 and B2. The smaller B1 the larger the increment (up to the capacity of the cache). Conversely, on a hit on B2, the smaller B2 the larger is the decrement (p lower bound is 0). If the size of the two lists could not be compared (either of the two the is 0) the increment is set to 1 by default.

In Table 5.5 we can see the working example of ARC; each column represents a step of the algorithm corresponding to a sector request (visible atop of each column) and each row represents a particular status of the corresponding ARC list (Table 5.5a). Each ARC list is a comma separated sequence of sectors, with the head of the list (mru position) represented by the leftmost number, while the tail of the list (lru position) is the rightmost number. In this example we suppose a cache of size three as in the previous examples. In Table 5.5b the first three steps of the algorithm are depicted; when a sector is referenced is placed at the head of list $T1$. The parameter p that is the desired $T1$ list size, is set to 0 at the beginning. When the cache is full and a new request arrives (table 5.5c), the lru sector of list $T1$ is evicted and sector 2 is placed at the mru position of list $T1$. The next reference (table 5.5d) is a hit and sector 0 is moved to the mru position of $T2$. The successive request (sector 3) is a miss (table 5.5e), sector 3 is placed in the mru position of list $T1$ and, since the list's size exceed the parameter p, ARC evicts the lru sector of $T1$ (and place a reference at the head position of list $B2$). Request for sector 3 is a hit (table 5.5f) and the sector is moved to the mru position of list T2. Successive request (table 5.5g) is a miss and sector 4 is placed in the mru position of list $T1$ with sector 2 evicted from the cache and referenced in the head position of $B1$. Since the size of lists $T1 + B1$ contains a number of sectors that equals the total size of the cache, the sector referenced in the lru position of $B1$ is removed. The next request (table 5.5h) is a miss, because sector 2 is neither in list $T1$ nor in list $T2$, but it is referenced in list $B1$. ARC places this sector to the head position of $T2$ and increases the target size p of list $T1$ by one ($p = 1$), then it removes the lru sector of list $T1$ because it exceed the target size p. The steps performed for the successive request are the same (table 5.5j), but this time the size of list $T1$ is less than p ($p = 2$) and the sector is evicted from list $T2$; the former lru sector of $T2$ (that was sector 0) is placed (as a reference) in the mru position of list $B2$. The request depicted in table 5.5k, shows a hit to list $B2$ (but a miss for the cache). In this case the parameter p is decreased by one ($p = 1$), the requested sector is placed in the mru position of list $T2$ and the lru sector of $T1$ is evicted (a reference to the evicted sector is placed in the mru position of list $B1$). The two successive requests (tables 5.5l and 5.5m) are two hits that simply move the requested sectors to the mru position of list $T2$. Request for sector 4 (table 5.5n) that is referenced in list $B1$ has the effect to increase the target

Cache references:

	7	0	1	2	0	3	0	4	2
					(∗)		(∗)		
T1	7	0,7	1,0,7	2,1,0	2,1	3,2	3,2	4,3	4
B1						1	1	2	3
T2					0	0	0	0	2,0
B2									
	(a)		(b)		(c)	(d)	(e)	(f)	(g)	(h)

	3	0	3	2	1	2	0	1
			(∗)	(∗)		(∗)		
T1	4				1	1		1
B1		4	4	4			1	
T2	3,2	0,3,2	3,0,2	2,3,0	2,3	2,3	0,2,3	0,2
B2	0				0	0		3
	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)	(q)

Table 5.5: ARC example

size p by one ($p = 2$) and place the sector in the mru position of $T1$. This time the candidate for eviction is the lru sector of list $T2$. Request for sector 2 (table 5.5o) is a hit to the list $T2$. Sector 0 in the successive request (table 5.5o) is referenced in list $T2$ and parameter p is decreased by one ($p = 1$); sector 0 is placed in the mru position of list $T2$. Finally the last request is for sector 1 that is referenced in list $B1$. In this case the sector chosen for eviction is the one in lru position of $T2$ and the requested sector is placed in the mru position of $T1$.

CAR

The problem of lru list management (like for example the lists of ARC) is that on a hit the cache page must be moved to the mru position of the list. In a typical multithreaded environment, the cache is protected by a lock to ensure consistency and correctness, that cause a great amount of contention. Moreover, with large caches, the overhead of continually moving the accessed sector to the mru position of the list, on every hit, is not acceptable. CAR is inspired by ARC and it tries to avoid the problem of a typical lru list based implementation described above, while maintaining the performance and adaptation mechanisms provided by ARC. It is based on the CLOCK algorithm that is an approximation of LRU that eliminates lock contention.

In Figure 5.4 we can see the detailed algorithm for CAR. In the next paragraphs we will give an overview of CAR and we will describe the steps of the algorithm when dealing with the sample list of requests (Table 5.1). The algorithm design is similar to that of ARC. There are still the lists T1, B1, T2 and B2 with the same meaning of the ARC policy. The difference with the aforementioned algorithm is that it maintains a strict lru ordering in lists T1 and T2, while CAR uses a one-bit approximation to LRU. As in ARC the CAR policy continuously uses the extra information history contained in lists B1 and B2 to adapt the size of T1 and consequently T2 (using the parameter p). The list T1 may contain sectors that have the reference bit set either to 1 or 0. The sectors inside list T1 with the reference bit set to 1 could be ideally grouped in a list, called $T1^{'}$ for reference. The sectors with reference bit are thus ideally grouped in list $T1$ $T1^{'}$. The cache replacement policy remove a sector from T1 if $T1 \setminus T1^{'}$ is greater than p otherwise it removes a sector from $T1^{'} \cup T2$. The adaptation rule is basically the same as ARC, while the cache replacement rule evict only the sectors with reference bit set to 0.

In Table 5.6 we can see the CAR algorithm applied to the working example (with a cache of size 3). In this example each row of each table represents a list of the algorithm (table 5.6a). Each list is referenced from left to right, meaning that the leftmost number in each cell represents the head of the list, while the tail of the list is the rightmost number. The number in parenthesis near each sector is the sector reference bit and the parameter p is initialized to 0 (the symbol $(*)$ indicates a cache hit). The first three steps of the algorithm are depicted in table 5.6b. At the beginning all lists are empty and a cache miss implies storing the requested sector in head position of list $T1$. After filling the cache, the request for the next sector (table 5.6c) requires a replacement by looking at the reference bit of the head sector of list $T1$ (because $T1$ size is greater than the parameter p). The evicted sector is not placed in $B1$ because the size of $T1$ plus $T2$ equals the maximum cache size. The next request is a hit, because sector 0 is already in $T1$, and consequently its reference bit is set to 1 (table 5.6d). The request for next sector (sector 3 in table 5.6e) is a miss, but this time the evicted sector is not placed in a history list, since its reference bit is one it is placed in list $T2$ (with reference bit set to 0). The successive sector in list $T1$ (sector 1) is demoted to the head of list $B1$ and the requested sector is placed in list $T1$. Successive request is a hit, sector 0 is in $T2$, and the algorithm set the reference bit for that sector to 1 (table 5.6g). Sector 4, requested next, is not in cache and thus is placed in list $T1$ with the consequence that sector 1 is removed from $B1$ and sector 2 is demoted from $T1$ and placed in $B1$ (table 5.6h). The next sector requested is sector 2 that is present in history list $B1$, this information

Cache references:

	7	0	1	2	0 (*)	3
T1	7(0)	7(0),0(0)	7(0),0(0),1(0)	0(0),1(0),2(0)	0(1),1(0),2(0)	2(0),3(0)
B1						1(0)
T2						0(0)
B2						
	(a)	(b)		(c)	(d)	(e)

	0 (*)	4	2	3	0 (*)
T1	2(0),3(0)	3(0),4(0)	4(0)		
B1	1(0)	2(0)	3(0)	4(0)	4(0)
T2	0(1)	0(1)	0(1),2(0)	0(1),2(0),3(0)	0(1),2(0),3(0)
B2					
	(f)	(g)	(h)	(i)	(j)	(k)

	3 (*)	2 (*)	1	2 (*)	0	1 (*)
T1			1(0)	1(0)	1(0)	1(1)
B1	4(0)	4(0)	4(0)	4(0)	4(0)	4(0)
T2	0(1),2(0),3(1)	0(1),2(1),3(1)	2(1),3(1)	2(1),3(1)	3(1),0(0)	3(1),0(0)
B2			0(0)	0(0)	2(0)	2(0)
	(l)	(m)	(n)	(o)	(p)	(q)	(r)

Table 5.6: CAR example

is used to increase the target size parameter p ($p = 1$) and successively move the sector to list $T2$ (table 5.6i). The successive request triggers the same action leaving list $T1$ empty (table 5.6j). The next request (for sector 0) is a hit and leaves the cache unchanged, because the sector reference bit is already equal to 1 (table 5.6k). The next two requests are two more hits, sector 3 and 2 are in list $T2$ and their respective reference bit is set to 1 (tables 5.6m and 5.6n). Successively, the miss for sector 1 has the effect to demote the head sector of list $T2$ to list $B2$; the requested sector is then placed in list $T1$ (table 5.6o). Request for sector 2 leaves the cache unchanged (table 5.6p). The next request, for sector 0, is a miss for the cache but a hit for the reference list $B2$ with the effect that the requested sector is swapped with the head sector of list $T2$ (table 5.6q); in this case the parameter p is decreased. Finally, sector 1 is already in the cache and its reference bit is set to 1 (table 5.6r).

5.1.3 Implicit Cache prefetching algorithms

Demand caching is the traditional assumption used for studying caching algorithms, where a sector is brought into cache only on a miss. Prefetching, on the other hand, speculates on which sectors will be accessed in the near future and selectively preload sectors in the cache before they are requested.

Several algorithms are known in the literature to exploit prefetching. The choice of which object prefetch is based on the prediction of likely future references, given an history of past accesses. The algorithms differ in the accuracy of the prediction mechanisms. A possible approach to increase accuracy is mining past history, but, whereas it is deeply studied subject, it has been rarely used in commercial systems, due to the complexity of the algorithms (that introduce a computational overhead) and for the necessity of long historic data to make the prediction sufficiently accurate.

For these reasons the most widely adopted algorithm exploits sequentiality of access, that is the request of a sector will be likely followed by request for subsequent sectors. The algorithms exploit sequential pattern of accesses first by detecting it and then by retrieving adjacent sectors from the main memory and put them into cache.

The simplest algorithm for sequential prefetching is synchronous prefetching. This algorithm merely load p extra sector in cache on a sector s request miss, thus loading into the cache the sectors s through $s+p$. The sector miss is interpreted as a sequential miss, that is, we are in presence of a sequential pattern but we have not already loaded all the sectors of that sequence in the cache. Although really simple this mechanisms is widely used in the context of prefetching because it has a low overhead and it is simple to be

implemented.

In order to increase the number of hits a more useful approach is to load sectors into cache, even without the detection of a sequential pattern. Typically when a special object (called trigger) is requested. The trigger tells the algorithm to prefetch successive objects even if a sequential miss is not occurred.

The prefetching mechanisms used in ENIGMA are: synchronous, synchronous plus asynchronous and SARC [38]. In the next sections we will review these algorithms.

Synchronous

As already anticipated in the previous paragraph, the simplest method of prefetching is synchronous prefetching. On every request for a sector s that represent a *miss* (i.e. sector s is not in the cache), a prefetch request is made for p successive sectors (beyond s); that is the proxy issues an increase redundancy requests for the sectors s through $s + p$ that are not already in the cache (and thus have a "normal" level of redundancy). Conversely if a sector exceed the cache boundaries, it is evicted from the cache and its redundancy is decreased to the standard level. The list of sectors that form the cache is managed as a LRU list.

Synchronous plus asynchronous

As previously noted, other prefetching mechanisms tries to adapt to different sequential patterns. This strategy, as the name suggests, tries to combine synchronous and asynchronous mechanisms. In the first instance the algorithm detects a sequential pattern by associating to each cached sector an identifier, namely "sequential count". On a read miss, if sector $s - 1$ is in cache and its sequential count exceed the threshold, a sequential miss is detected and the proxy sends an increase redundancy command to the successive p sectors (and a reference to those sectors is stored in the cache). Along this request a sector is marked as trigger, that if accessed will produce subsequent asynchronous prefetch requests. The trigger is the last sector in the prefetch request minus a prefixed parameter g.

SARC

As described in [38], SARC tries to capture both temporal and spatial locality in a single prefetching mechanisms. SARC partitions the cache space between random data (for temporal locality) and sequential data (for spatial locality). The algorithm starts with a given cache partitioning and dynamically tweaks

it to find the optimum. In practice a desired cache size is calculated for both the sequential data and random data, using the marginal utility of allocating more space to the sequential or random prefetched data. At any given step if the marginal utility of sequential list is higher than that of random list, the desired size is increased, otherwise is decreased.

SARC is inspired by ARC and maintains two lists, namely RANDOM and SEQ. The policy dynamically adapts the amount of space given to sequential data (SEQ list) and random data (RANDOM list). The goal of the algorithm is to avoid cache pollution, that is when requested sectors are evicted from the cache in favor of prefetched sectors(speculatively chosen and possibly less precious). As in ARC the desired SEQ list size is given by the parameter p. The sectors are evicted from the lru end of list SEQ if its size is greater than p, otherwise the sector are evicted from the lru end of list RANDOM. The adaptation of the parameter p is performed by calculating the marginal utility of allocating more space to list SEQ and list RANDOM, that is how the misses experienced by a list change as the list size change. If at a certain time the marginal utility of SEQ is higher than that of RANDOM, the parameter p is increased, otherwise is decreased. In Figure 5.5 we can see the detailed SARC algorithm.

In the next section we will describe the results obtained by ENIGMA using the architecture and caching algorithms discussed in the previous sections.

```
INITIALIZATION: Set p = 0 and set the lists T1, B1, T2, and B2 to empty.

CAR( x)
INPUT: The requested page x.
1:  if (x is in T1 ∪ T2) then /* cache hit */
2:      Set the page reference bit for x to one.
3:  else /* cache miss */
4:      if (|T1| + |T2| = c) then
            /* cache full, replace a page from cache */
5:          replace()
            /* cache directory replacement */
6:          if ((x is not in B1 ∪ B2) and (|T1| + |B1| = c)) then
7:              Discard the LRU page in B1.
8:          elseif ((|T1| + |T2| + |B1| + |B2| = 2c) and (x is not in B1 ∪ B2)) then
9:              Discard the LRU page in B2.
10:         endif
11:     endif
        /* cache directory miss */
12:     if (x is not in B1 ∪ B2) then
13:         Insert x at the tail of T1. Set the page reference bit of x to 0.
        /* cache directory hit */
14:     elseif (x is in B1) then
15:         Adapt: Increase the target size for the list T1 as: p = min {p + max{1, |B2|/|B1|}, c}
16:         Move x at the tail of T2. Set the page reference bit of x to 0.
        /* cache directory hit */
17:     else /* x must be in B2 */
18:         Adapt: Decrease the target size for the list T1 as: p = max {p − max{1, |B1|/|B2|}, 0}
19:         Move x at the tail of T2. Set the page reference bit of x to 0.
20:     endif
21: endif
```

```
replace()
22: found = 0
23: repeat
24:     if (|T1| >= max(1, p)) then
25:         if (the page reference bit of head page in T1 is 0) then
26:             found = 1;
27:             Demote the head page in T1 and make it the MRU page in B1.
28:         else
29:             Set the page reference bit of head page in T1 to 0, and make it the tail page in T2.
30:         endif
31:     else
32:         if (the page reference bit of head page in T2 is 0), then
33:             found = 1;
34:             Demote the head page in T2 and make it the MRU page in B2.
35:         else
36:             Set the page reference bit of head page in T2 to 0, and make it the tail page in T2.
37:         endif
38:     endif
39: until (found)
```

Figure 5.4: CAR algorithm

INITIALIZATION: Set the adaptation variables to 0.

```
1:  Set seqMiss to 0
2:  Set adapt to 0
3:  Set desiredSeqListSize to 0
```

CACHE MANAGEMENT POLICY:

Track x is requested:

```
4:  Set ratio = (2 · seqMiss · ΔL)/seqListSize

5:  case i: x ∈ RANDOM (HIT)
6:      if x ∈ RANDOM BOTTOM then
7:          Reset seqMiss = 0
8:          Set adapt = max( -1, min(ratio - 1, 1) )
9:      endif
10:     Mru(x, RANDOM)

11: case ii: x ∈ SEQ (HIT)
12:     if x ∈ SEQ BOTTOM then
13:         if (ratio > LargeRatio) then
14:             Set adapt = 1
15:         endif
16:     endif
17:     if x is AsyncTrigger then
18:         ReadAndMru([x + 1, x + m − x%g], SEQ)
19:     endif
20:     Mru(x, SEQ)
21:     if track (x − 1) ∈ (SEQ ⋃ RANDOM) then
22:         if (seqCounter(x − 1) == 0) then
23:             Set seqCounter(x) = max (seqThreshold,
                    seqCounter(x − 1) + 1)
24:         endif
25:     else
26:         Set seqCounter(x) = 1
27:     endif

28: case iii: x ∉ (SEQ ⋃ RANDOM) (MISS)
29:     if (x − 1) ∈ (SEQ ⋃ RANDOM) then
30:         if seqCounter(x − 1) == seqThreshold then
31:             seqMiss++
32:             ReadAndMru([x, x + m − x%g], SEQ)
33:             Set seqCounter(x) = seqThreshold
34:         else
35:             ReadAndMru([x, x], RANDOM)
36:             Set seqCounter(x) =
                    seqCounter(x − 1) + 1
37:         endif
38:     else
39:         Set seqCounter(x) = 1
40:     endif
```

CACHE MANAGEMENT POLICY (CONTINUED):

```
ReadAndMru( [start, end], listType )
41: foreach track t in [start, end]; do
42:     if t ∉ (SEQ ⋃ RANDOM) then
43:         grab a free track from FreeQ
44          read track t from disk
45:     endif
46:     Mru(t, listType)
47: done
48: if (listType == SEQ)
49:     Set AsyncTrigger as (end − triggerOffset)
50: endif
```

FREE QUEUE MANAGEMENT:

```
FreeQThread()
51: while (true) do
52:     if length(FreeQ) < FreeQThreshold then
53:         if (seqListSize < ΔL
            or randomListSize < ΔL) then
54:             if (lru track of SEQ is older than
                lru track of RANDOM) then
55:                 EvictLruTrackAndAdapt(SEQ)
56:             else
57:                 EvictLruTrackAndAdapt(RANDOM)
58:             endif
59:         else
60:             if (seqListSize > desiredSeqListSize) then
61:                 EvictLruTrackAndAdapt(SEQ)
62:             else
63:                 EvictLruTrackAndAdapt(RANDOM)
64:             endif
65:         endif
66:     endif
67: endwhile

EvictLruTrackAndAdapt( listType )
68: evict lru track in listType and add it to FreeQ
69: if (desiredSeqListSize > 0) then
70:     Set desiredSeqListSize += adapt / 2
71: else
72:     Set desiredSeqListSize = seqListSize
73: endif
```

Figure 5.5: SARC algorithm

5.2 Caching policies evaluation

In this section we provide an evaluation of the caching algorithms described previously. In the next section we review the methodology used and we comment the results.

We evaluated the caching architecture of ENIGMA with regard to the latency of sectors retrieval. The goal of applying caching algorithms to ENIGMA is to reduce average access time (latency). The key features of our simulation software are described in section 4.3.1. In this section and the next we will describe simulation settings and we will evaluate results. Then, we will choose the best policy for EC and IC based on these results. We will also evaluate how performance are related to caches dimension and to the amount of increase in sector redundancy level. The evaluation is performed feeding the simulator with real workload traces, as described in section 4.3.1.

In the next section we will describe the scenario and settings in detail, then we will comment the simulation results.

5.2.1 Scenario

The two caches are evaluated with a suitable combination of these algorithms. Each policy determines the choice for the sector to be replaced (if the cache is full). If a particular policy is applied to EC the sector is evicted from the cache. Conversely if a policy is applied to IC, the redundancy of the sector evicted is decreased up to a standard level. Conversely, a sector in EC has faster access time, whereas a sector in IC has an increased level of redundancy (that reduces access time).

ENIGMA borrows some similarities from existing storage systems where the algorithms described previously are applied, but it has also some peculiarities on its own. As pointed out in [72] (whereas in a different context), in a multilevel cache environment, if the temporal locality (i.e. the working set of most accessed sector) is completely exploited by the first level cache, accesses to the second level cache are actually misses from the first level (i.e. the pattern of access to the second level cache does not follow the temporal locality principle). For this reason the algorithms for EC and IC should differ in order to prevent poor IC utilization.

In a standard disk the latency for retrieving a sector is dominated by the time needed by the head to seek the correct cylinder. Since this operation is slow, the disk controller tries to gain advantage by reading all the trace rather than a single sector. Given the absence of mechanical components in ENIGMA (whose positioning would take time), such strategies are not feasible. In practice in order to take advantage of prefetching and exploit spatial

locality, we have to apply other strategies. A possible criterion is to use IC as a prefetching cache, in order to exploit spatial locality of requests. The EC cache, on the other hand, has the task of capturing temporal locality. In the next section we will outline the settings used to perform the simulations; in section 5.2.3 we will evaluate the EC algorithms when no IC cache is present, then in section 5.2.4 we will evaluate IC prefetching algorithms when no EC cache is present. Finally in section 5.2.5 we will evaluate the cache when used together.

5.2.2 Simulation settings

We choose a sector size of $4KB$ which is a standard size of modern large capacity disk drives. Following table 4.1, we choose a fixed value of k and n, that is $k = 128$ and $n = 256$. Fragments are placed uniformly at random over the storage nodes of the infrastructure.

We experimented several EC cache dimension ranging from $64MByte$ to $1GByte$, because modern large size hard disk drive have typically $64MByte$ cache value and because the proxy has to store caches of several users. The IC cache instead has less space constraints because it relies on the storage capacity of the infrastructure. Thus we used larger values for the IC cache: $1GByte$ and $2GByte$.

The redundancy level determines how fast a sector is retrieved. As already mentioned, placing a sector in the IC cache means that its redundancy is increased; in the experiments we pick several values for the amount of redundancy applied each time, in order to study how redundancy affects performance. The values are a percentage of the k fragments needed to reconstruct the sector and are added to the total amount of fragments n. The percentage selected are 10%, 20%, 50%, 100% and 150%.

5.2.3 EC cache policy evaluation

In this section we describe the results obtained by using the caching algorithms applied to EC cache against the workload discussed in 4.3.1, varying cache size and without using the IC cache. In figures 5.6, 5.8, 5.10, 5.12 we can see, on the x-axes, the cache size in MByte whereas, on the y-axes, the reduction in percentage of the average latency for retrieving a sector. In Figures 5.7, 5.9, 5.11, 5.13 we evaluate the cache hits in percentage over the total number of requests.

CFS workload (4.3.1)

In figures 5.6 and 5.7 we can see the performance of the various caching algorithms, when the simulation is driven by the CFS workload. CFS workload processes several small size requests (few sectors at the time) with short inter arrival time. It is the workload with the higher number of unique file accesses, which means that, most of the request for a given sector are made only once. Moreover the number of the most requested files is very high, which means that the temporal locality is divided between a large number of files.

Proportionately, the maximum increment of performance is obtained with a small value of cache size; $64MByte$ succeed in reducing the average latency of the 20%. This improvement is proportional to the number of cache hits. Increasing the cache size further does not produce an increment as we could expect. The relationship between the cache size and the number of hits is non-linear; doubling the cache size does not halve the percentage of cache hits respect to the previous value. Moreover, for cache size values above $512MByte$ the increment of performance is not large.

As we can see from Figure 5.6, the performance of all the caching algorithms are quite similar. The only algorithm whose performance is not satisfactory is LRU, because it is too simple and its replacing policy dos not cope with the complexity of this workload.

The reasons for these results is due to the particular behavior of the workload. Most of the time the workload requests different files (with few sector requests per file) and all the requests arrive within small time interval. The result of this behavior is that the cache is accessed frequently, with an high degree of concurrency between the requests, which makes the sectors stay in cache for shorter time. The higher number of requests, most of which are unique, and the high I/O rate increases the probability that popular sectors are evicted from the cache shortly after they are pushed into (to make space to new requested sectors) and before they are accessed again.

Given the unsteadiness of the workload, the algorithms that seek to adapt to the workload (namely ARC and CAR) performs similar to those algorithm with fixed parameters (namely 2Q and LIRS). In this case the adaptation fails to improve performance because it is impossible to adapt to the continuous variation of the workload.

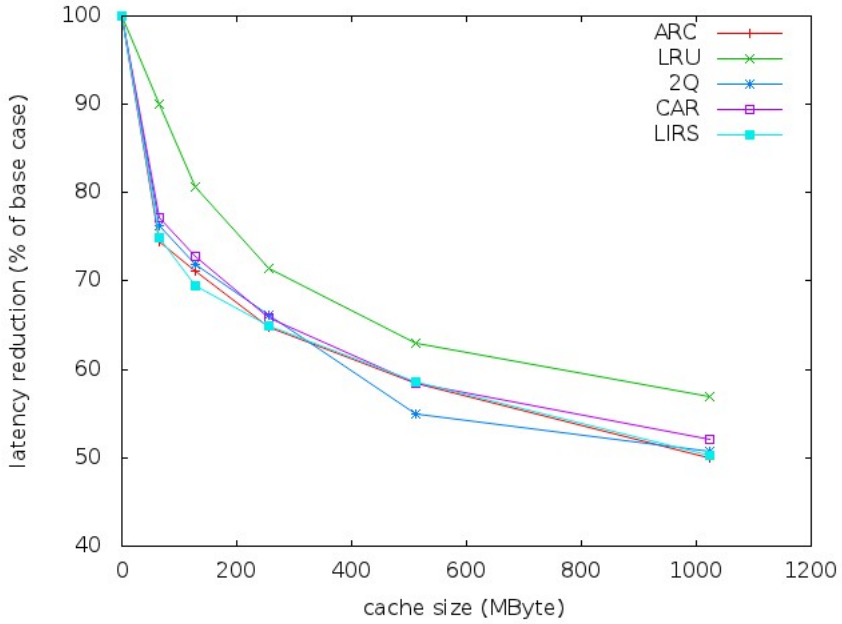

Figure 5.6: CFS workload with several EC cache size and algorithms. Average Latency 95% confidence interval and 2.5% relative error.

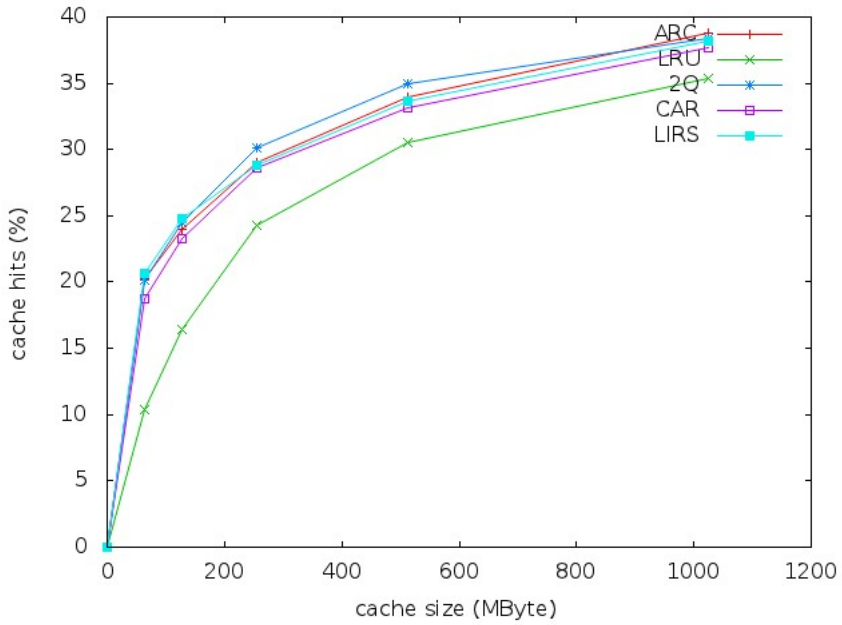

Figure 5.7: CFS workload with several EC cache size and algorithms. Cache hits (%)

DAP-DS workload (4.3.1)

In figures 5.8 and 5.9 we can see the performance of the various caching algorithms when the simulation is driven by the DAP-DS workload. DAP-DS workload has a low I/O rate, most of the requests arrive within one second time interval and are directed to a single file (i.e. the sectors requested are always the same). the requests have a medium-size sequentiality (that is, each requests is made for a series of contiguous sectors, in this case 8 contiguous sectors on average).

As we can see in Figure 5.8 and 5.9, unlike the previous workload, the relationship between cache size and performance benefits is linear, meaning that increasing the size of the cache results in a proportional increase of cache hits (and consequently latency reduction). This could be explained by the fact that I/O request rate is low and thus the sectors remain for a longer period in the cache. This is helpful because this workload requests more frequently the same file and hence its sectors have the chance of remain in cache and be accessed again for each request. The file is probably large and it is partially stored in the cache. The more the cache size increase, the more the sectors of the file are stored. In particular, increasing the size of the cache proportionally increases the number of sectors contemporaneously stored and allows the algorithm to choose the worst sector for eviction.

With this workload the algorithm with higher adaptation characteristics, namely ARC, performs better. The adaptation allows the algorithm to maintain in cache the set of sectors that will be requested again with higher probability; this set is continuously adapted to the changing characteristics of the workload. The other algorithms, however, are really close to the best one, except for LRU that performs badly for every cache size (it fails to maintain the most accessed sectors in cache).

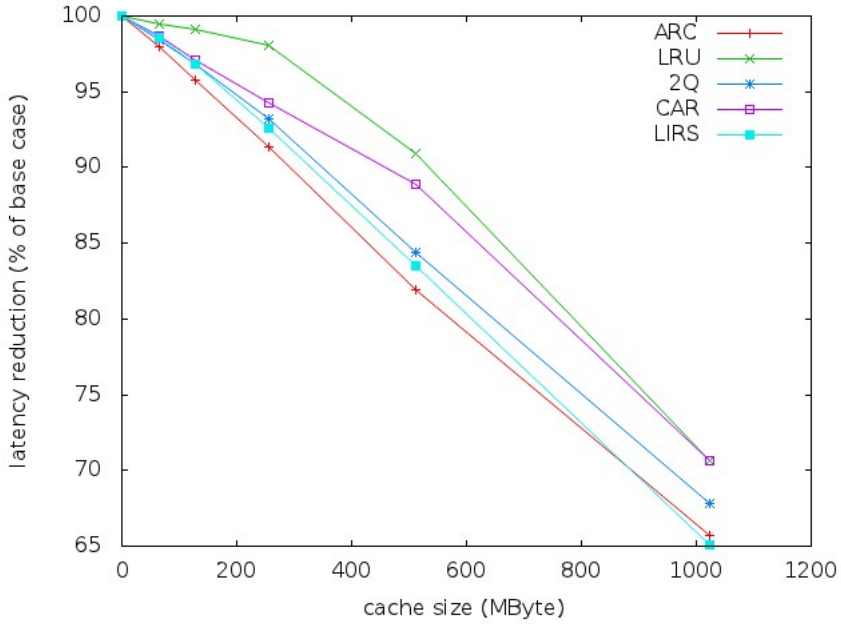

Figure 5.8: DAP-DS workload with several EC cache size and algorithms. Average Latency 95% confidence interval and 2.5% relative error.

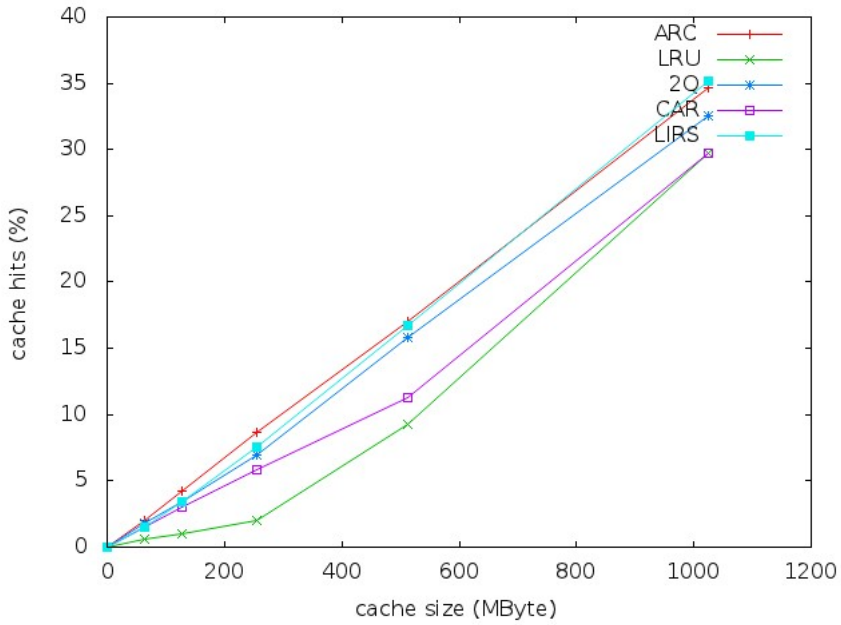

Figure 5.9: DAP-DS workload with several EC cache size and algorithms. Cache hits (%)

WBS workload (4.3.1)

In figures 5.10 and 5.11 we can see the performance of the various caching algorithms when the simulation is fed by the WBS workload. The average size of the requests equals the size of the sector, that is, most of the requests are of $4KByte$ size. Moreover, 13% of the requests are purely sequential, meaning that several contiguous sectors are requested at the same time. The number of files requested is very high and corresponds to a poor correlation between successive requests (a small number sectors is requested two or more times)

As we can see in Figure 5.11 a small cache size can hold the most accessed sectors, indeed with a cache of size $64MByte$ we have an number of hits that is around 15% of the requests, whereas successive increments of the cache size increase the number of hits of only 2%. This behavior, however, does not provide a significant performance improvement, because the most frequently requested sectors are far less than the total amount of sectors requested. Most of the requests are sequential or are directed to a wide number of sectors, requested only once.

As we can see in Figure 5.10, there are no significant differences between the algorithms. In particular LRU performs as well as the other algorithms and even better in some cases.

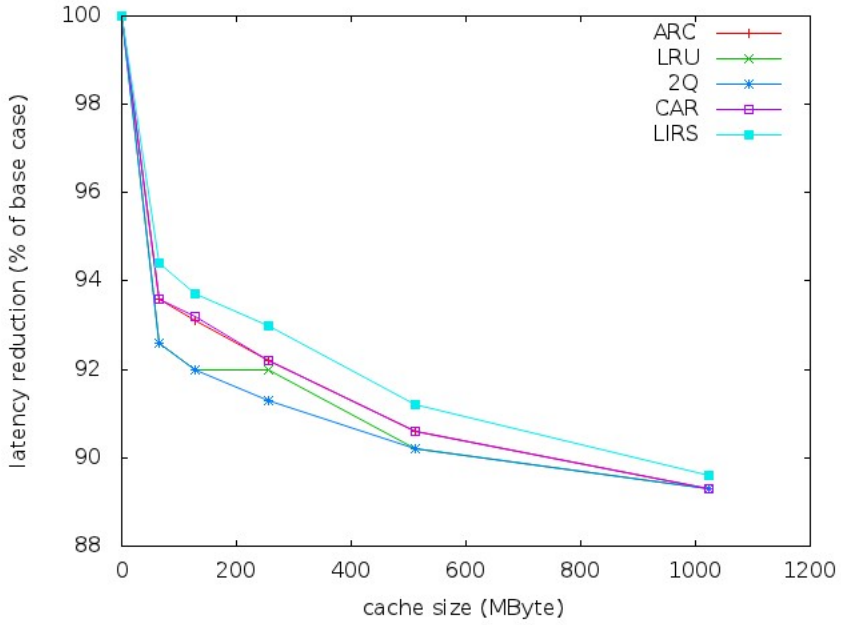

Figure 5.10: WBS workload with several EC cache size and algorithms. Average Latency 95% confidence interval and 2.5% relative error.

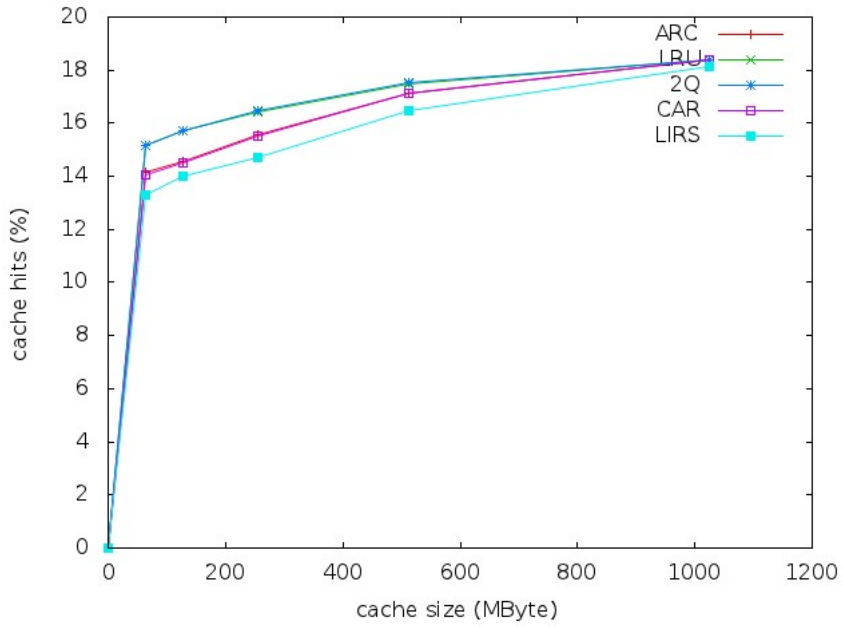

Figure 5.11: WBS workload with several EC cache size and algorithms. Cache hits (%)

RAD-BE workload (4.3.1)

In figures 5.12 and 5.13 we can see the performance of the various caching algorithms when the simulation is driven by the RAD-BE workload. This workload represents the best testbed for ARC, because it has strong temporal locality and high I/O rate. The 80% of requests are directed to two files and consequently a subset of the total number of sectors (that corresponds to the sectors that form the file) are requested most of the times.

As we can see in Figure 5.13 the graph is quite similar to that of the previous workload (5.11), but in this case the number of hits is two times higher on average, leading to a decrement of latency of almost half the value compared to the experiments where no cache is present. This could be explained by noting that the higher degree of temporal locality is exploited even for smaller cache size ($64MByte$ cache suffice to hold the most requested sectors), and this workload does not have sequential requests that could pollute the cache with useless sectors.

As we can see in 5.12 ARC outperforms the other algorithms for every cache size. This is due to its higher adaptation characteristics that performs particularly well with this workload.

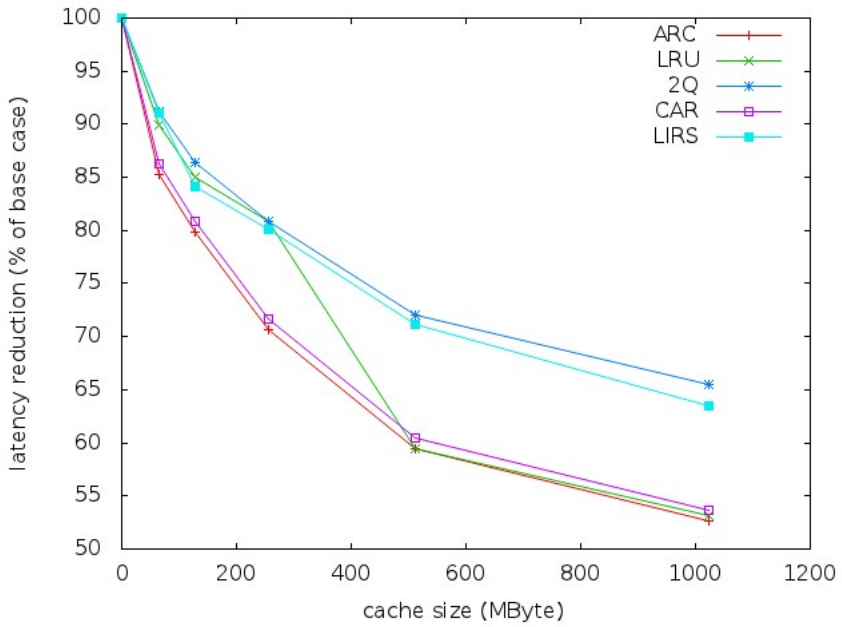

Figure 5.12: RAD-BE workload with several EC cache size and algorithms. Average Latency 95% confidence interval and 2.5% relative error.

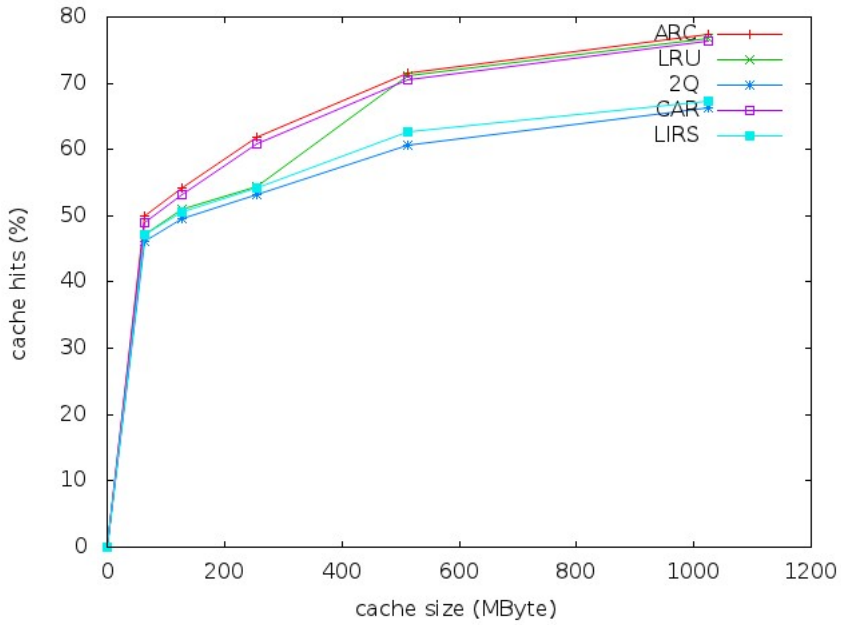

Figure 5.13: RAD-BE workload with several EC cache size and algorithms. Cache hits (%)

5.2.4 IC cache policy evaluation

In this section we provide an evaluation of the IC cache alone, in order to establish the best amount of redundancy to give to each sector present in the cache at a particular moment. We also evaluate the IC cache algorithms and the size of the IC cache. The workloads are the same of the previous set of experiments. We compare each choice with regard to the average latency of retrieving a sector.

As we can see from figures 5.14, 5.15, 5.16 and 5.17 the performance improvement due to IC prefetching algorithms is less variable than for the EC caching algorithms.

Given this results we can conclude that the IC size does not improve performance, meaning that there is no significant difference between the performance of ENIGMA with a cache of $1GByte$ size and a cache of $2GByte$ size. Moreover the choice of the prefetching algorithm is not decisive, there is no significant difference between the algorithms in most of the cases. These results could be explained by noting that the sequential requests of each workload are not frequent and the cache is probably polluted most of the time with useless sectors. The constant factor in the reduction of the average latency, that is similar for every workload (20% of reduction of latency), may suggest that, although sequentiality is not completely exploited, some of the sectors in IC cache are still randomly requested.

The redundancy increment is effective only for values between 60% and 100%. For smaller values, the average latency does not decrease significantly. In this case if we add new fragments to the total number of fragments that compose the sector, it is unlikely that they will contribute significantly to decreasing the latency if their number is far less than the total number of fragments. In particular, if we add few fragments to a sector that is composed of hundred of fragments, the new ones contribution will be hidden by other fragments (that are far more).

For values over 100% the average latency does not change (for different number of fragments this threshold would be different). This could be probably explained by noting that, when the number of fragments become comparable to the number of nodes of the infrastructure, the probability of storing new fragments on faster nodes decrease (most of the nodes have been already used for storing a fragment). In practice, when the number of fragments exceeds a certain threshold, it is as if we have already made the most of all network resources.

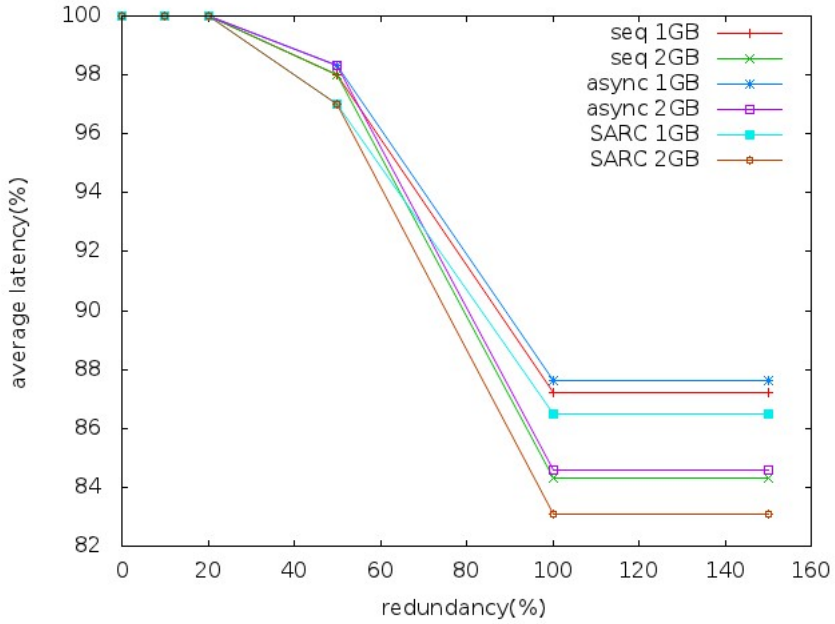

Figure 5.14: CFS workload varying IC cache size and redundancy. Average latency 95% confidence interval and 2.5% relative error.

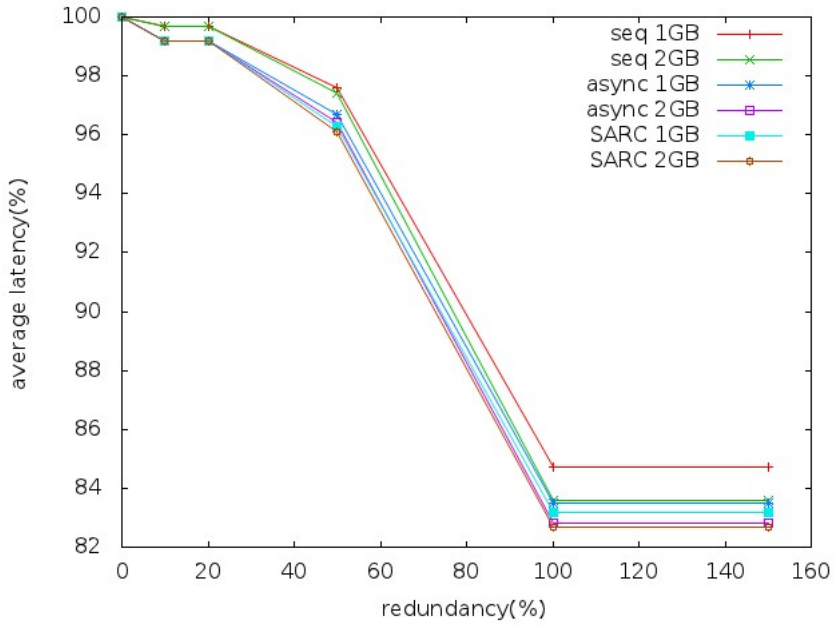

Figure 5.15: DAP-DS workload varying IC cache size and redundancy. Average latency 95% confidence interval and 2.5% relative error.

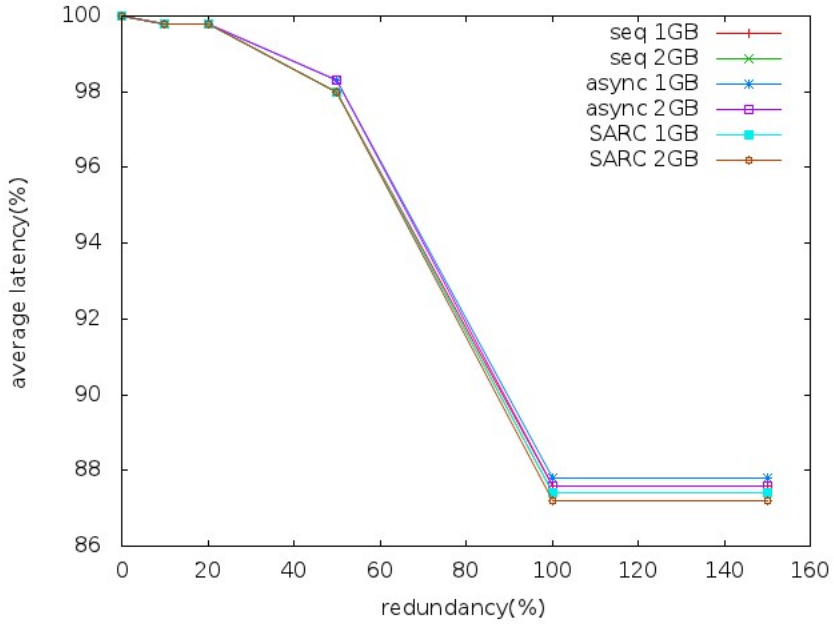

Figure 5.16: WBS workload varying IC cache size and redundancy. Average latency 95% confidence interval and 2.5% relative error.

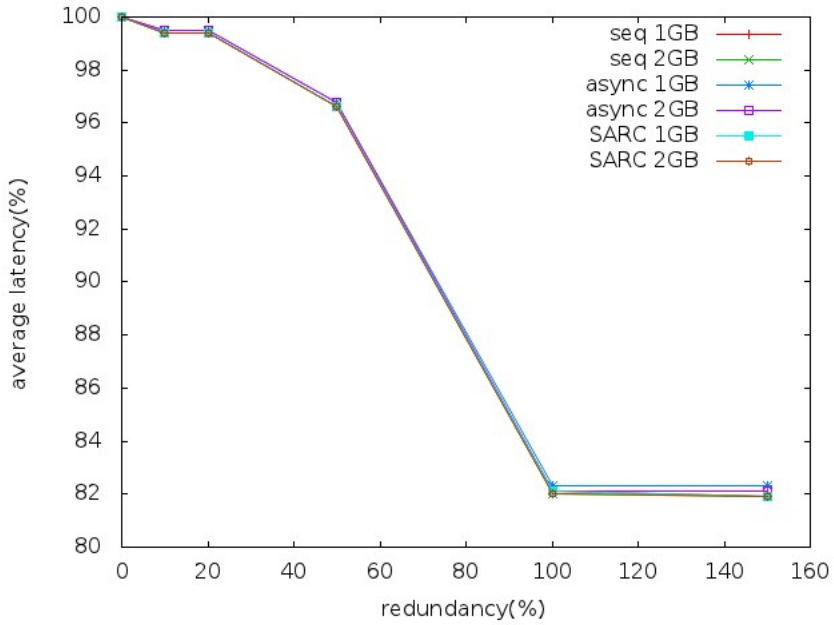

Figure 5.17: RAD-BE workload varying IC cache size and redundancy. Average latency 95% confidence interval and 2.5% relative error.

5.2.5 Combination of EC and IC algorithms

In this section we compare the combination of EC and IC algorithms to find one that could suites ENIGMA needs. We choose to compare the caching algorithms ARC and 2Q that are representative of two classes of algorithms, respectively, algorithms with an higher degree of adaptability and algorithms with tunable parameters. We combined each algorithm with two prefetching algorithms for IC cache described previously, namely seq and SARC, that represents respectively a simple (but widely used) algorithm and an adaptive algorithm. We choose to use an EC cache of size $256MByte$ and an IC cache of size $1GByte$ with an increasing amount of redundancy. In figures 5.18, 5.19. 5.21 and 5.20 we can see the average sector retrieval time in millisecond. The constant functions represented by the lines titled "2Q no IC cache" and "ARC no IC cache" are the latency of retrieval with IC cache disabled (only EC cache is present). The top line, named "no cache" is the average latency when either cache are off line. As we can see in the pictures the combination of both caches (EC cache and IC cache) can lead to significant performance improvement.

In Figure 5.18 we can see that the contribution of EC cache in decreasing the latency is greater than the contribution of IC cache. This could be explained by noting that the CFS workload has poor locality and most of that locality is exploited by EC cache. The EC algorithm does not alter significantly the result, meaning that, as noted previously, with this kind of workload all algorithms have the same performance.

In Figure 5.19, conversely, the contribution of IC cache is greater, because the DAP-DS workload has a sequential components in the stream of requests. As we can see, the IC cache contribution to the reduction of latency is similar with respect to the EC contribution. With this particular workload, the more sophisticate algorithm, namely ARC and SARC, performs better, because of their ability to adapt to the varying workload characteristics.

As we can see in Figure 5.20, also in this case IC cache contribution to performance improvement is comparable to that of EC cache. Most of the WBS requests are sequential and the remaining requests are unique accesses, meaning that the corresponding fragments are requested only once. Given this dual nature of the workload the SARC algorithm performs better compared to the simple prefetching scheme, due to its ability to cope with both sequentially accessed data and randomly accessed data.

In Figure 5.21 the algorithms are tested using the RAD-BE workload. This workload has a strong temporal locality, meaning that most of the time the same few files are requested. For this reason the EC cache provides most of the benefits to performance improvement.

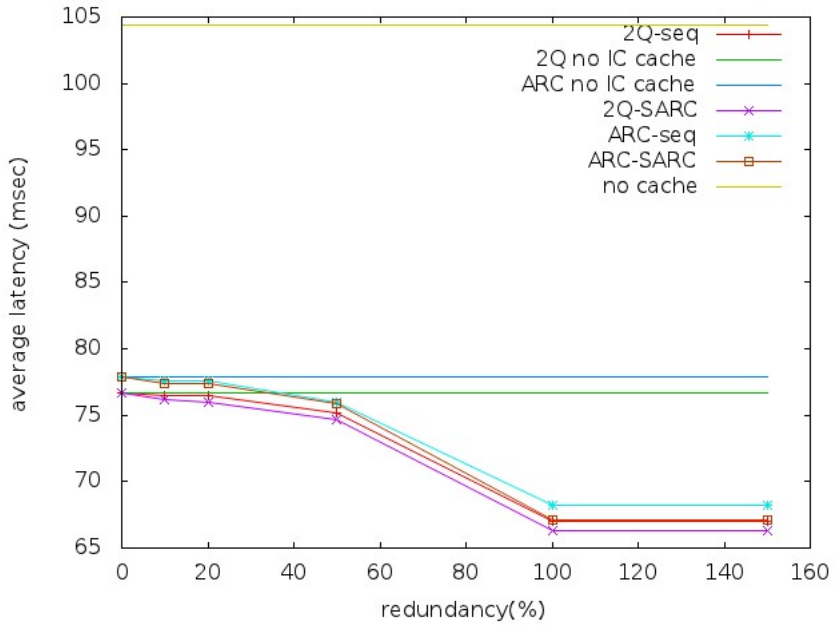

Figure 5.18: CFS workload with EC and IC. Average Latency 95% confidence interval and 2.5% relative error.

In this chapter we provided a review of caching algorithms used in ENIGMA and we studied, by means of simulation techniques, how these algorithm performs in ENIGMA. As we can see from the previous discussions, using caching algorithms brings considerable benefits. Unfortunately, the algorithms that work well in traditional systems, do not show significant differences when applied to ENIGMA. In the next chapter we will make concluding remarks and we will give some hints for possible future works.

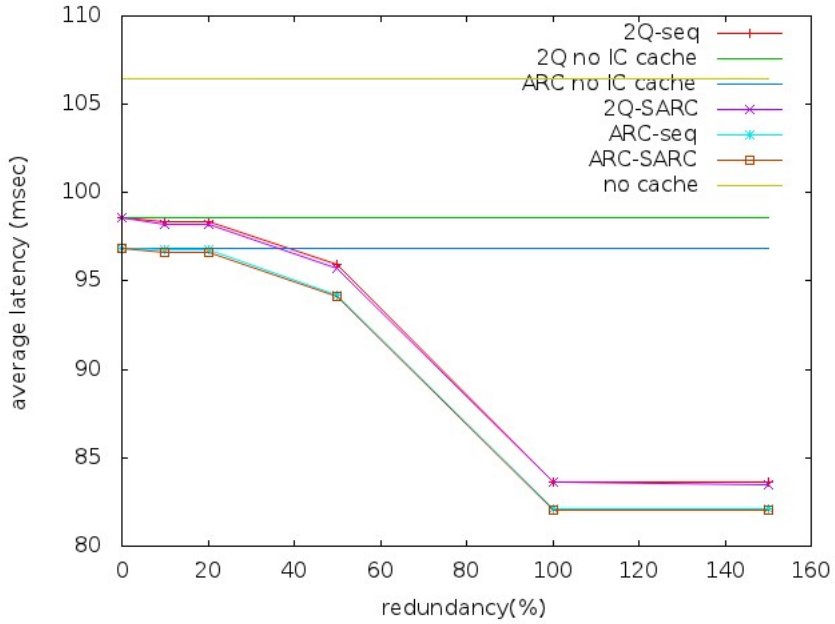

Figure 5.19: DAP-DS workload with EC and IC. Average Latency 95% confidence interval and 2.5% relative error.

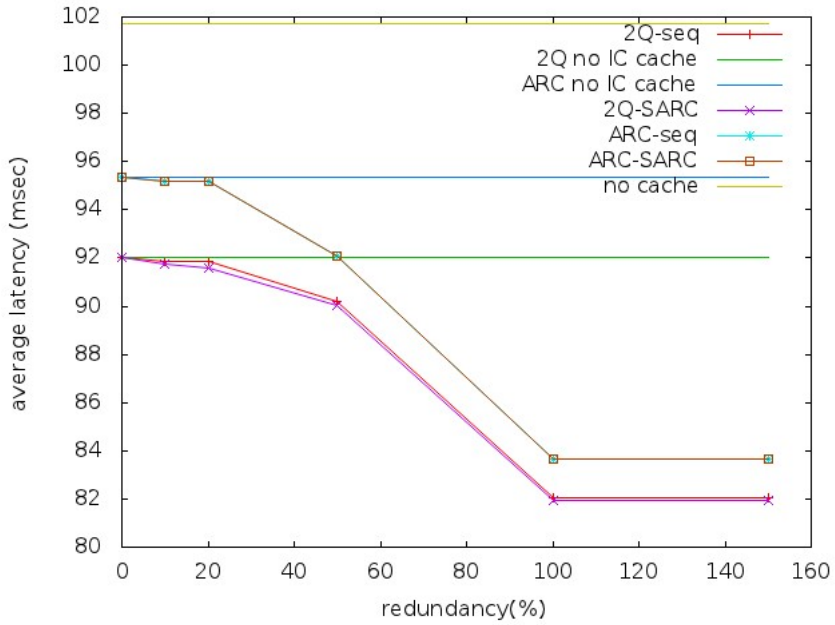

Figure 5.20: WBS workload with EC and IC. Average Latency 95% confidence interval and 2.5% relative error.

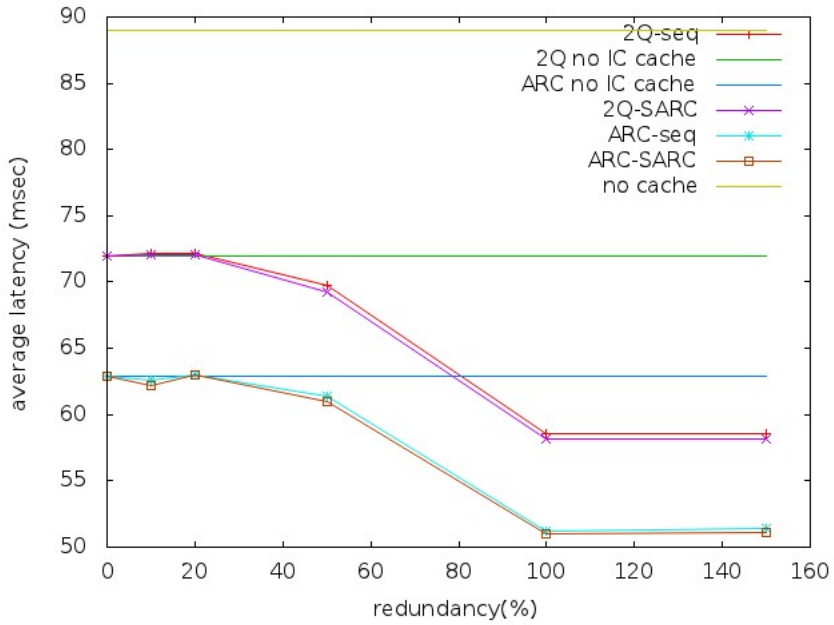

Figure 5.21: RAD-BE workload with EC and IC. Average Latency 95% confidence interval and 2.5% relative error.

Chapter 6

Conclusions and future work

In this thesis we proposed ENIGMA, a distributed infrastructure that provides virtual disks that can be used either directly by the VMs hosted on a Cloud infrastructure, or as the back-end for VBD systems.

ENIGMA exploits erasure coding techniques to encode each sector of a virtual disk as a set of fragments independently stored on a set of physical storage nodes to achieve tunable large storage capacity, high availability, strong confidentiality, and high data access performance. In particular, we used LT erasure codes, with the addition of a second level of coding in order to preserve confidentiality when increasing and decreasing the redundancy level of given sectors of a virtual disk.

The architecture of ENIGMA is based on proxy nodes that coordinate the usage of storage nodes providing access to their local storage to store encoded sector fragments. Storage nodes are organized in clusters and each cluster is coordinated by a cluster-head. The set of cluster-heads forms a logical peer-to-peer network with arbitrary topology.

The caching architecture of ENIGMA is organized in two levels; at first level, a traditional cache is located in the proxy and stores recently used sectors, that will be likely reused in the near future. The second level of caching uses the ability of erasure coding techniques to increase the redundancy level of selected sectors in order to decrease sectors retrieval time (conversely the proxy can decrease the redundancy level of sectors that are no longer requested). This special cache is located in the infrastructure and it is composed with all the sectors with increased redundancy; the proxy stores only a reference to those sectors, not the sectors themselves. The second level of caching is used for prefetching purposes, that is, to increase in advance the redundancy level of sectors that will be likely requested in the future. Caching and prefetching techniques are used to decrease average disk access time performance.

ENIGMA is resilient to attacks by a single malicious storage node, attempting to decode fragments of disk sectors as well as by a more sophisticated attacker that has gained sufficient knowledge of the distributed storage system to retrieve a set of K coded fragments of a particular sector. Furthermore, it is possible to derive design criteria to obtain a desired availability level, based on analytical models developed specifically for ENIGMA.

ENIGMA obtains high data access performance by letting the proxy manage overlapped operations, by simultaneously fetching many fragments of the same sector, and by dynamically increasing or decreasing sectors redundancy (this last feature provides also the basic mechanism that can be used to keep virtual disk performance at a given level in face of migration of the VM accessing it).

The metrics used for evaluating ENIGMA virtual disks performance are average sector retrieval time and overall throughput and are obtained with a ad-hoc simulator of the ENIGMA system.

ENIGMA disk throughput is evaluated using several values of proxy bandwidth. The results shows that the infrastructure can provide an aggregate throughput that is adequate to exploit the full proxy channel (for reasonable values of proxy bandwidth). Sector retrieval latency is analyzed using different caching and prefetching algorithms. ENIGMA caching infrastructure provides a general reduction in the average sectors access time; increasing redundancy produces benefits, in term of access time reduction, that grows up rapidly for initial increments, then, when the number of fragments becomes comparable to the total number of resources, increasing the redundancy level further on, does not bring adequate advantages.

Simulation was used also to test ENIGMA tolerance to failures. In the simulations performed, the storage nodes have a non-zero probability of failure and, when an appropriate redundancy level is chosen, ENIGMA can tolerate failures with little performance degradation.

ENIGMA characteristics are different compared to other storage solutions.

Commercial storage solutions, like for example Dropbox, provide a simple and transparent way to integrate a cloud storage with the client file system. Unlike ENIGMA it offers a file system like interface to access data. In this case a software has to be developed for each operating system in order to interact with the Dropbox. ENIGMA, on the contrary, provides access at a low level and it relies on standard software like, for example, iSCSI). The storage capacity provided by Dropbox is flexible, ranging from few GByte up to 100GByte; ENIGMA storage capacity is flexible as well, but with the limit given by the aggregate storage capacity of the infrastructure (that could be potentially in the order of TeraBytes). Unlike ENIGMA, Dropbox

is able to work offline, but we think that this is a minor difference, because the amount of redundancy of a virtual disk can cope with network failures as well. One of the peculiarities of the Dropbox is that it allows folder sharing between users (users can access the same data). ENIGMA does not directly provide this feature, however it could be used as a raw device with a software layer atop of it, that provides concurrent access. In order to provide confidentiality, Dropbox uses standard encryption mechanisms (e.g., ssh and AES), that could possibly introduce computational overhead. The use of LT codes in ENIGMA provides the basic mechanisms for confidentiality at a low computational cost. Finally, Dropbox performance could be an issue, because it relies on remote data centers to provide its functionality and has no means of improve performance, as can do ENIGMA.

ENIGMA performance, respect to a traditional storage solution (e.g., SAN), is clearly not comparable. However there are situation in which a distributed system like ENIGMA is preferable. For example ENIGMA compared to a SAN, can increase and decrease the storage capacity without buying new hardware. In practice storage capacity is added on demand on a pay-per-use basis. Storage space offered by existing solution is getting larger due to the increasing demand for it, but in the context of SAN or storage arrays the availability is bound to the availability of a single datacenter. ENIGMA on the contrary uses redundancy (and in particular LT codes) to provide availability in spite of failure; moreover availability can be tuned at runtime, to meet the demand of the user (stipulated for example with SLA).

For these reasons ENIGMA has intermediate characteristics compared to cloud and traditional storage solutions. With regards to existing back up solution, for example, it has better access time, because it was designed to be used in more performance sensitive scenario. Moreover ENIGMA can provide high availability, a key feature for back up systems. If used as a disk, ENIGMA access time is higher than that of traditional storage, but it is comparable with the access time of virtual disks used by virtual machines. However, compared to virtual disks ENIGMA offers greater availability and the ability to cope with virtual machine migration (existing virtual disk are accessible only inside a datacenter).

6.1 Future work

In this section we will sketch future directions we would like to investigate in order to further increase ENIGMA capabilities.

In this Thesis we performed an analysis, with simulation techniques, of read access time performance of a cloud disk provided by ENIGMA; in order

to evaluate the virtual disk in all its aspects, we plan to investigate the performance in case of writing as well; in particular we would like to test if the mechanisms adopted by ENIGMA suffice to guarantee an adequate level of consistency and writing throughput.

The second level of coding (introduced to preserve confidentiality) does not exploit the full potential of erasure codes and introduces a management overhead. Using the second level of coding has the drawback of increasing the information that the proxy must store and process, and it also introduces a new kind of encoded fragment that does not have the strong erasure coding properties of LT codes. This means that the more we use the second level of coding, the more new (second level) encoded fragments are difficult to be created (because of linear dependencies between each second level encoded fragment). Moreover, traditional network codes for distributed storage [31], unlike LT codes, takes into account also the repair bandwidth. When a node fails or reliability must be increased a new fragment must be created using original information. This operation involves reconstructing a given sector and creating new fragments, an operation that only the proxy can do, with the drawback of increasing computational complexity and bandwidth usage. Regenerating codes were created to face the problem of error repair. A key feature of these codes is that they are able to minimize the bandwidth necessary to repair a failure. We plan to investigate the feasibility of using LT features to solve both problems: eliminate the need of a second level of coding and cut down repair bandwidth. To the best of our knowledge, using LT codes for addressing the repair problem ([13]), is still an open problem of research in this area.

As we showed, dynamically increasing and decreasing sectors redundancy level is a valuable mechanism to improve access time performance. We also showed that the benefits obtained by this technique increase up to a certain limits and then further increasing redundancy does not longer reduce access time. We plan to study other techniques to improve performance. In particular one of the basic assumptions of ENIGMA is that each fragment is placed uniformly at random over the infrastructure, that is, each storage node has the same probability of storing a fragment of a given sector. We plan to investigate how performance varies when different placement policies are used. This problem is somewhat similar to the distributed service placement problem, present in the literature (for example in [61]). Distributed service placement algorithms aim to find a suitable placement of a service inside a network, given a cost metric (for example CPU, available bandwidth and available storage).

The placement of fragments over the infrastructure could cause an imbalance of the load on the network, with heavily loaded nodes and, consequently,

a degradation of performance. In practice a fragments placement that is near-optimal at a certain time could become non-optimal in the future. We think that could be useful to use techniques for integrating fragment placement with mechanisms that periodically restore the balance of the network and the load on the storage nodes. We plan to study the feasibility of applying data migration algorithms known in the literature (for example in [18] and [33]) to ENIGMA, also from the perspective of improving performance in face of virtual machines migration.

In articles [64], [62] and [63], the authors conducted a series of studies that address problems similar to that of ENIGMA, whereas in the context of peer-to-peer backup storage. In the articles the authors addresses the problem of how data placement and bandwidth allocation affects the performance metric, that is the time required to complete a backup. Moreover they studied the problem of how scheduling policy adopted for uploading large data files in presence of churn (in the peers) affects the performance metric. We think that techniques and formal studies exposed in the aforementioned articles could be adopted in the context of ENIGMA and we plan to investigate their applicability in the near future.

At the moment we are working on an implementation of ENIGMA, in order to test its behavior in a real environment and compare simulated results with real measurement.

Bibliography

[1] Retrieved on Nov. 22^{nd}, 2011.

[2] Amazon Elastic Block Storage. Retrieved on Nov. 22^{nd}, 2010.

[3] Amazon Elastic Compute Cloud (Amazon EC2). Retrieved on Nov. 22^{nd}, 2010.

[4] box. Retrieved on Nov. 8^{nd}, 2011.

[5] Dropbox. Retrieved on Nov. 8^{nd}, 2011.

[6] FP7 Vision Cloud. Retrieved on Feb. 18^{nd}, 2012.

[7] Google App Engine. Retrieved on Nov. 8^{nd}, 2011.

[8] Jungle Disk. Retrieved on Nov. 8^{nd}, 2011.

[9] Kernel Based Virtual Machine. Retrieved on Nov. 22^{nd}, 2010.

[10] PlanetLab. Retrieved on Nov. 22^{nd}, 2010.

[11] SNIA IOTTA Repository. "Retrieved on Sept. 19^{th}, 2011".

[12] The Amazon Simple Storage Service (Amazon S3). Retrieved on Nov. 22^{nd}, 2010.

[13] The Repair Problem. Retrieved on Nov. 8^{nd}, 2011.

[14] VMware. Retrieved on Nov. 22^{nd}, 2010.

[15] Windows Azure. Retrieved on Nov. 8^{nd}, 2011.

[16] Windows Live Mesh. Retrieved on Nov. 8^{nd}, 2011.

[17] Michael Abd-El-Malek, William V. Courtright, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: versatile cluster-based storage. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, page 5. USENIX Association, 2005.

[18] Eric Anderson, Joseph Hall, Jason D. Hartline, M. Hobbes, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. Algorithms for data migration. *Algorithmica*, 57(2):349–380, 2010.

[19] Volker Kuhn Andre Neubauer, Jurgen Freudenberger. *Coding theory: algorithms, architectures, and applications.* John Wiley and Sons, 2007.

[20] S. Banerjee, B. Bhattacharjeey, and C. Kommareddy. Scalable application layer multicast. In *Proc. of 2002 SIGCOMM Conference.* ACM, October 2002.

[21] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '09)*, Shangai, China, May 2009. IEEE CS Press.

[23] Adam L. Beberg and Vijay S. P. Storage@home: Petascale distributed storage. In *In IPDPS*, pages 1–6. IEEE, 2007.

[24] V. Bioglio, R. Geata, M. Grangetto, and M. Sereno. On the fly gaussian elimination for lt codes. *IEEE Communication Letters*, 13:953–955, 2009.

[25] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, March 2004.

[26] R. Buyya, R. Ranjan, and N. Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *Proc. of 10th International Conference on Algorithms and*

Architectures for Parallel Processing (ICA3PP), Busan, South Korea, May 2010. Springer, Germany.

[27] John W. Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: using tornado codes tospeed up downloads. In *in INFOCOM 99. Eighteenth AnnualJoint Conference oftheIEEEComputer andCommunications Societies. Proceedings.* IEEE, 1999.

[28] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley, New York, 1991.

[29] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Gs^3: a grid storage system with security features. *J. Grid Comput.*, 8(3):391–418, 2010.

[30] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM Press.

[31] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A Survey on Network Codes for Distributed Storage. *Proceedings of IEEE*, 99(3), March 2011.

[32] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared, 2008.

[33] Rajiv Gandhi and Julian Mestre. Combinatorial algorithms for data migration to minimize average completion time, 2010.

[34] Gregory R. Ganger, Pradeep K. Khosla, Mehmet Bakkaloglu, Michael W. Bigrigg, R. Garth, Semih Oguz, P. Vijay, Craig A. N. Soules, John D. Strunk, and Jay J. Wylie. Survivable storage systems. In *In DARPA Information Survivability Conference and Exposition, IEEE*, volume 2, pages 184–195, 2001.

[35] X. Gao, M. Lowe, and M. Pierce. Supporting Cloud Computing with the Virtual Block Store System. In *Proc. of 5^{th} IEEE International Conference on e-Science (e-Science '09)*, Oxford, UK, December 2009. IEEE CS Press.

[36] Roxana Geambasu, Amit Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key/value store. In *Proc. of OSDI*, 2010.

[37] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. of 19th ACM Symposium on Operating Systems Principles*, Lake George, NY, USA, October 2003.

[38] Binny S. Gill and Dharmendra S. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *In Proc. of USENIX 2005 Annual Technical Conference (2005*, pages 293–308, 2005.

[39] Yunhong Gu and Robert L. Grossman. Sector: A high performance wide area community data storage and sharing system. *Future Generation Comp. Syst.*, 26(5):720–728, 2010.

[40] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics, 3rd ed.* New York: Macmillan, 1970.

[41] Howie H. Huang, John F. Karpovich, and Andrew S. Grimshaw. Analyzing the feasibility of building a new mass storage system on distributed resources. *Concurrency and Computation: Practice and Experience*, 20(10):1131–1150, 2008.

[42] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Marina Del Rey*, pages 31–42. ACM Press, 2002.

[43] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[44] Sung ju Lee, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Rodrigo Fonseca. Measuring bandwidth between planetlab nodes. In *In PAM*, pages 292–305, 2005.

[45] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production Windows Servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128. IEEE, 2008.

[46] Ranjita Bhagwan Kiran, Kiran Tati, Yu chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *In NSDI*, pages 337–350, 2004.

[47] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the 7th international conference on Architectural support for programming languages and operating systems*, pages 84–92, New York, NY, USA, 1996. ACM.

[48] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure code replication revisited. In *In PTP04: 4th International Conference on Peer-to-Peer Computing. IEEE*, pages 90–97, 2004.

[49] M. Luby. LT codes. In *IEEE FOCS*, pages 271–280, November 2002.

[50] P. Mahajan, S. Setty, S. Lee, A. Seehra, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. of OSDI*, 2010.

[51] Petar Maymounkov and David Mazieres. Rateless codes and big downloads. In *In IPTPS'03*, 2003.

[52] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[53] Michael Mitzenmacher. Digital fountains: A survey and look forward, 2004.

[54] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Proc. of 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '09)*, Shangai, China, May 2009. IEEE CS Press.

[55] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype, 2003.

[56] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication.

[57] S. Lee S. Kim. Improved intermediate performance of rateless codes. *ICACT 2009*, 3:1682–1686, February 2009.

[58] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39:48–58, October 2004.

[59] S. Sanghavi. Intermediate performance of rateless codes. In *Information Theory Workshop*, pages 478–482, sept 2007.

[60] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, June 2006.

[61] Todd Sproull and Roger D. Chamberlain. Distributed algorithms for the placement of network services, 2008.

[62] Laszlo Toka, Matteo Dell'Amico, and Pietro Michiardi. On scheduling and redundancy for p2p backup. 09 2010.

[63] Laszlo Toka, Matteo Dell'Amico, and Pietro Michiardi. Online data backup : a peer-assisted approach. In *P2P'10, 10th IEEE International Conference on Peer-to-Peer Computing, August 25-27, 2010, Delft, The Netherlands*, 08 2010.

[64] Laszlo Toka, Matteo Dell'Amico, and Pietro Michiardi. Data transfer scheduling for p2p storage. In *P2P'11, IEEE International Conference on Peer-to-Peer Computing, August 31-September 2nd, 2011, Kyoto, Japan*, 08 2011.

[65] Francesco Tusa, Massimo Villari, and Antonio Puliafito. Credential management enforcement and secure data storage in glite. *IJDST*, 1(1):76–97, 2010.

[66] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven H. Parallax: Managing storage for a million machines. In *In Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[67] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *In Proceedings of the First International Workshop on Peer-to-Peer Systems IPTPS 2002*, 2002.

[68] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[69] B. Wong, A. Slivkins, and E. Gun Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *Proc. of 2005 SIGCOMM Conference*, Philadelphia, PY,USA, August 2005.

[70] Lamia Youseff, Maria Burtico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, 2008.

[71] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. 10.1007/s13174-010-0007-6.

[72] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 15(7):2004, 2004.

[73] Matteo Zola, Valerio Bioglio, Cosimo Anglano, Rossano Gaeta, Marco Grangetto, and Matteo Sereno. Enigma: Distributed virtual disks for cloud computing. *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, pages 898–906, 2011.