The Manna Plug-In Architecture for Content-based Search of VM Clouds

Wolfgang Richter, Glenn Ammons[†], Jan Harkes, Adam Goode,
Nilton Bila[‡], Eyal de Lara[‡], Mahadev Satyanarayanan

August 2010
CMU-CS-10-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]IBM Research, [‡]University of Toronto

This research was supported by the National Science Foundation (NSF) under grant number CNS-0614679, and by an IBM Open Collaborative Research grant. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, IBM, Carnegie Mellon University or the University of Toronto.

Keywords: data-intensive computing, virtual machines, cloud computing, interactive search, forensic analysis, computer vision, pattern recognition, distributed systems, Diamond, OpenDiamond, Mirage

Abstract

As cloud computing becomes more popular, collections of virtual machine (VM) images are growing in size. Management of VM collections requires the ability to inspect and search data stored within VM images. We present a plug-in-based architecture, called Manna, for efficiently searching state within VM images through both index and non-index based search. The architecture offers a flexible framework for creating a wide range of new applications that are valuable to both end users and administrators of VM images. We showcase this flexibility through three applications built using Manna's API: one for searching images, one for searching source code, and one for performing virus scanning. Efficient search for such diverse applications is achieved using two independent mechanisms: plug-ins that are data-type-specific, but independent of data source, and use of VM-specific metadata to shrink the search space of non-indexed data. Trace-driven measurements on our prototype confirm that Manna searches incur low performance overhead.

1 Introduction

Virtualization of systems is a growing trend. Virtualization encapsulates the execution state of a computer in a virtual machine (VM) image. We use the vendor-neutral and format-neutral term *parcel* to refer to such an image. A parcel is typically stored as a self-contained disk image within a compute cloud. The growing popularity of virtualization has led to a rapid growth of collections of parcels [11]. Users create snapshots of parcels for backup and failure recovery. Administrators create and deploy parcels with customized software packages for individual users. A recent paper [18] documents the growth of parcels within three clouds: IBM's Research Compute Cloud (RC2) [1], North Carolina State University's VCL cloud [22], and Carnegie Mellon University's Internet Suspend/Resume® cloud (ISR) [9, 17].

Managing large collections of parcels requires the ability to search data in parcels and to quickly identify relevant parcels for the task at hand. The following hypothetical scenarios reflect examples of queries users and system administrators may need to make on parcels:

- A team of graphic designers developing a marketing campaign for a travel agency wishes to aggregate all documents and images relevant to the current project that are scattered throughout the team's various parcels.
- A videographer is working on an old project using a codec that he hasn't used in a long time. He realizes, after a series of software updates, that the latest version of the codec contains a new bug not previously seen. He wants to identify and revert to the most recent parcel snapshot with a bug-free version of the codec.
- A system administrator needs to identify and patch parcels that contain outdated versions of a software package with recently published vulnerabilities.

In this paper, we explore *efficient search of cloud-based parcels*. We identify requirements for searching large collections of parcels efficiently and present *Manna,* a plug-in architecture for searching cloud-based parcels.

An obvious approach to supporting search on parcels is to boot each parcel individually and to use facilities available within the parcel's OS to index and search its files. Alternatively, to eliminate the need to power up each parcel individually and interface with a diverse set of search tools across parcels, an administrator could mount each parcels' file systems onto a single server and perform the search with a single tool on the server. However, both of these approaches are inefficient. Many parcels share much of the same content, including OS and software packages [18]. Indexing and searching parcels individually results in redundant processing of identical files. We address this inefficiency in two steps. First, we provide deduplication by replacing individual parcel disk images with a global content-addressable storage. Second, we build a global index enabling rapid search across parcels. Section 3 describes these steps in more detail.

While some searches can be efficiently answered with pre-built indices, others involve deep semantic knowledge and hence need to be performed interactively on parcel contents. For example, while the team of graphic designers can quickly locate all documents containing the term,"beach," via indices that have been built offline for all parcels, they would be unable to locate photographs that include beach scenes, but have not been explicitly tagged with that term.

For this open-ended class of deep semantic searches that involve parcel content and not just parcel metadata, Manna leverages the Diamond architecture for discard-based search [8, 19]. For example, graphic designers can express searches for photographs using primitives such as regions of specific colors or textures, human faces, animals, etc., rather than relying on low-level regular expressions that lack semantic value and do not lead to satisfactory results. In addition, Manna can support direct involvement of a human expert in directing all aspects of the search process, including many iterative refinements of the search query based on partial results. Section 4 describes these aspects of Manna in more detail.

In Sections 2 to 4 below, we discuss the challenges of searching cloud-based parcels, infer requirements, and describe the Manna architecture and prototype implementation. Then, in Section 5, we describe

three Manna applications: one to perform searches of unlabelled collections of images, another to search source code, and a third to search for virus and software vulnerabilities. In Section 6, we present experimental results that validate Manna's ability to perform parcel management searches with low performance overhead—typically within 3% of searches on data within local file systems.

2 Motivating Applications and Induced Requirements

Virus and vulnerability search is a critical system administration task that must be performed in order to guarantee a level of security within production parcels. To ensure that no parcels are infected, a virus scan must encompass every file in each parcel stored in any particular cloud. Because this type of search is exhaustive, it seems infeasible for large collections of parcels. However, the following insights make for a more efficient and feasible solution. First, although exhaustive, this problem can be partitioned by using multiple scanners to search files in parallel. Second, each unique file in the cloud should be scanned only once, and the time needed to run the scan is the time it takes to go through every unique file in the cloud. We know from previous work [18] that there is significant duplication of data in clouds. Using deduplication we can decrease the time needed to perform a scan by shrinking the search space. By combining deduplication with parallel scanning, the problem becomes tractable because we have shrunk the search space and can scale to larger collections of parcels by adding scanners. This virus and vulnerability scanning example induces three requirements: first, a mechanism for file-granularity indexing of parcels that is independent of parcel format, OS, and file system; second, a mechanism for deduplicating files; and third, a flexible method of providing data to a virus scanning engine and obtaining results.

For *image search*, consider the illustrative example of how a user might search for pictures of a wedding. The user begins her search using face detection, reasoning that wedding pictures are likely to contain faces. While the returned images contain faces, many of them are not from the wedding. So she adds a color constraint to match the wedding dress. Unfortunately, the color is not accurate and the new search returns no relevant images. The user then remembers that some of the wedding photos were taken outdoors. She replaces the wedding dress constraint with a new green color constraint to match grass. The new query returns a photo that includes the bride standing in front of the church. Based on the color of the bride's bouquet, the user creates another constraint that, together with the face criterion, produces a satisfactory set of images from the wedding. This image search example induces three new requirements: human-in-the-loop interactivity which supports iterative refinements of the search query, extensible plug-in architecture for complex constraint expression such as color and texture, and the ability to narrow searches based on metadata—Manna need not consider text files, for example, when searching for images.

Source code search has many potential uses, including searching for deprecated programming practices, locating snippets of potentially vulnerable source code, searching for copyrighted code, and general source-based analysis. With enough context, static analyzers could provide the criterion for a search—or entire compiler packages such as LLVM [10]. The level of abstraction when searching source code varies from the low level interpretation of raw bytes, to high level representations such as tokens or parse trees. If the abstraction is raw bytes, or some encoding such as UTF-8, then source code search degenerates into pattern matching. Effective source code search induces one new requirement: an ability to provide data context that is distinct from user context. If this requirement is not met, the problem becomes intractable for a user because source code files may depend on the transitive closure of a large number of other files. This is a requirement we are currently implementing in Manna.

3 Indexed Search of VM Clouds

Today, index-based search of structured data is a well-understood problem with a wide range of solutions from industry and academia that have resulted in well-documented best practices [7]. There are two main problems, mentioned already in the virus and vulnerability example of Section 2, in applying this extensive

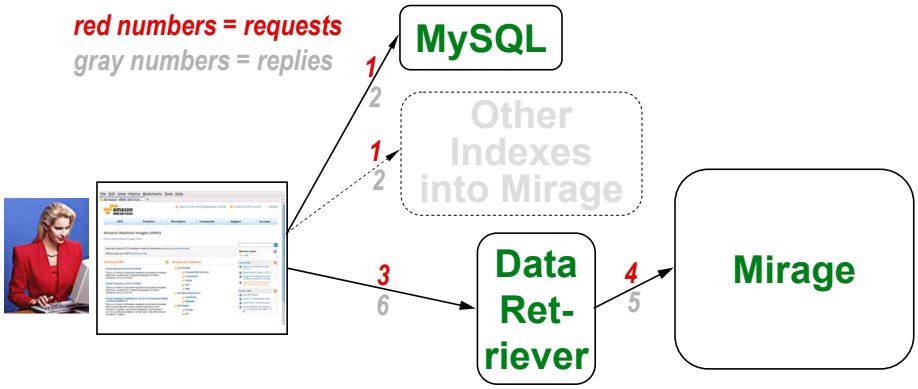


Figure 1: Indexed Search Architecture

prior work to cloud-based parcels. First, the data to be indexed is buried inside parcels and is therefore not directly accessible to existing search engines. Second, there is a need for deduplication of parcel content in a cloud before indexing. We elaborate on these issues below.

The first problem we face is that the data to be searched by Manna is not directly accessible, but is embedded within parcels whose internal formats have to be parsed. In a large cloud, there may be considerable diversity of parcel formats because of different virtual machine monitors (VMMs), guest operating systems, and file system types. One user may use parcels based on VMware Workstation 7.0 and encapsulating a Fedora 12 guest environment with an `ext4` file system. Another developer may use parcels based on the latest version of KVM and a Fedora 10 guest environment with an `ext3` file system. Coping with the large cross-product of VMMs, guest environments, and file system formats is a challenge. We could index based on raw blocks, but we lose access to the rich semantic metadata provided by the OS and filesystem of a given parcel. We would also have no method of indexing on a per file basis.

The second problem we face is taking advantage of the considerable similarity across parcels [18, 13], typically due to use of the same guest OS and applications. For example, if the parcels of two users both use Windows XP Service Pack 3 as the guest, there are likely to be many executable and system configuration files in common across the two parcels. As another example, many files may be unchanged in snapshots of the same user's parcel many days or weeks apart. Factoring out identical data across parcels is important for two distinct reasons. There is the obvious storage savings on servers through deduplication. In addition, only one copy of a file that is identical across parcels needs to be examined in a search. The cumulative savings over time from the latter can be huge, since the payoff occurs on every search that involves copies of that file. Exploiting similarity at a semantic level (such as file contents) is likely to be more satisfactory than at a low level (such as 4KB disk blocks) because of differences in parcel sizes, layouts, formats, and histories.

We describe Manna's solutions to these problems in Sections 3.1 to 3.3 below. We begin with an architectural overview in Section 3.1 and then look deeper into specific mechanisms in the following sections.

3.1 Indexed Search Architecture

Figure 1 shows how Manna supports index-based search. In this figure, *Mirage* is a parcel storage system from IBM that was designed to reduce VM sprawl in large clouds [16]. The object labeled *"Data Retriever"* in this figure is a data abstraction component of Manna that hides details of data location, layout, format and access methods from higher levels of the system.

Prior to the search, structured data is extracted from parcels in Mirage and indexed. The results of these actions can be stored in the MySQL database as well as in auxiliary index data structures from systems like

BigTable [5]. For example, MySQL may contain the pathnames and inode information of all files from all parcels in Mirage, while a separate full-text index is created of the subset of those files that are text files.

The message flows shown in Figure 1 provide an illustrative example of how index-based search works in Manna. Using a client GUI, a user interacts with the MySQL database and any additional indexing structures (1). These sources return the pathnames of files that match the user's query (2). Using the returned list, the client presents these pathnames to the data retriever (3). The data retriever looks up the files within Mirage (4), loads the files (5), and finally returns them to the client (6).

3.2 Parcel Analysis and Deduplication

Manna uses a combination of Mirage and MySQL to store the contents of parcels and the metadata extracted from them. The simpler approach of using Mirage alone is not satisfactory. This is because the external interfaces to Mirage were designed primarily to store and retrieve parcels, and are not expressive enough for complex file-specific metadata queries.

Provisioning of Manna with new parcels occurs in a number of steps. In the first step, a preprocessing tool transforms a parcel from its VMM-specific format (such as the `vmdk` format used by VMware) into the raw disk image format expected by Mirage. In the second step, Mirage imports the parcel by examining partitions specified by the user. Since Mirage does not currently support parcels with memory images, guest OSes have to be shut down rather than suspended to create an acceptable parcel. Mirage is currently able to parse Linux file system formats such as `ext2` and `ext3`, as well as the Windows NTFS format. From each partition, Mirage extracts the directory structure, file/directory names and attributes, and file contents. The contents of each file are copied into a content-addressable persistent store, using the SHA-1 hash of the file as its unique tag. Deduplication of file contents occurs trivially at this step. This aspect of Mirage is similar to that of Venti [15], including the use of storage arenas. In the third step, a postprocessing tool extracts the metadata for the parcel from Mirage and inserts it into MySQL. Note that the tool does not copy file contents into MySQL. Instead, the SHA-1 field of a file tuple serves as a backpointer to its content in Mirage.

The MySQL organization follows directly from the main abstractions of Manna. There is a parcel table keyed by a unique parcel id. Each parcel has an entry in this table with fields corresponding to its metadata such as owner, creation date, platform type, and size. There is a file table, with one entry for every file in every parcel. The fields of this entry include the unique id of the parcel to which the file belongs, pathname of the file, inode information such as owner and mode bits, and the SHA-1 hash of the file contents. Although contents are deduplicated, there is a distinct entry for every file in a parcel. Finally, there is a file location table for use by the data retriever mechanism that is described in the next section.

3.3 External Data Retrieval

A data retriever encapsulates the details of external data sources. An implementation of this abstraction has to provide a method for retrieving the contents of specified objects. A planned extension to this abstraction will allow a list of object identifiers to be provided initially, followed by an iterator-style `GetNextObject()` call to obtain the contents of the next available object from that list. This provides data sources with an important degree of freedom in reordering data accesses for optimal performance. For example, a data retriever could return cached objects first, and overlap network accesses for uncached objects with the processing of earlier objects by Manna servers. This would reduce stalls in the Manna pipeline from the data source to the user.

The data retriever for Mirage is implemented using standard Web technology, since this has been highly optimized for data throughput and request service rate. We use a `lighttpd` server, which is able to sustain roughly 1400 Mirage requests per second on our hardware. The data retriever methods use the fields of the MySQL database to map Mirage object identifiers to pathnames and offsets in the local file system where the contents of files extracted by Mirage are stored.

4 Searching Non-Indexed Data in VM Clouds

Since a VM encapsulates arbitrary computing state, some content searches may be at a semantic level that is too deep for current indexing technology. The difficult challenges of indexing for queries of high semantic content have long been investigated by the knowledge retrieval community [12, 24]. While steady progress continues to be made, there have been no breakthrough advances to date. For the foreseeable future, Manna will need to support such queries by directly involving a human-in-the-loop. For example, consider the following query: "In the C source code files in parcels created by developers from the XYZ project in the past week, are there any that have this bad programming practice?" File types, file locations, project affiliations, and timestamps represent structured information that can easily be indexed. Deprecated programming practice, on the other hand, is a much fuzzier concept that is best handled through a mechanism that directly involves a human-in-the-loop.

To handle deep semantic search of complex non-indexed data, Manna leverages a technique called *discard-based search* [8, 19] whose essence is query-specific content-based computation pipelined with human cognition. The query-specific parallel computation shrinks a search task down to human scale, thus allowing the expertise, judgement and intuition of an expert to directly influence the specificity and selectivity of the search. We first provide background on discard-based search in Section 4.1. Then, in Section 4.2, we describe an extension to this mechanism called *scoping* that narrows the search space by using structured data such as the MySQL-Mirage combination shown in Figure 1. Finally, we present the complete Manna architecture in Section 4.3.

4.1 Background: Discard-based Search

The *OpenDiamond® platform* is an open source implementation of Linux middleware for discard-based search [14]. Its extensible architecture cleanly separates the domain-specific and domain-independent aspects of the problem. User interaction typically occurs through a domain-specific GUI. The domain-specific code that performs early discard on servers is called a *searchlet.* It is typically composed of individual components Manna calls *plug-ins.* For example, an image search application may provide plug-ins for face detection, color detection, and texture detection. Plug-ins embody specific dimensions of knowledge, while searchlets express a multi-dimensional search query as a precedence graph of parametrized plug-ins. Two optimizations are used to ensure an adequate rate of return of results to an interactive user. First, the OpenDiamond workload is embarrassingly parallel, making it easy to add more servers to speed up searches. Second, temporal locality in search queries is exploited by persistent caching of search results at servers. This can be viewed as a form of *just-in-time indexing* that occurs incrementally as a transparent side-effect of user activity.

A searchlet is composed and presented by the application through the Searchlet API, and is distributed to all of the servers involved in the search task. Each server iterates through the locally-stored objects in a system-determined order and presents them to plug-ins for evaluation through the Plug-in API. Each plug-in can independently discard an object. A server is ignorant of the details of plug-in evaluation, only caring about the scalar return value that is thresholded to determine whether a given object should be discarded or passed to the next plug-in. Only those objects that pass through the entire gauntlet of plug-ins in the searchlet are transmitted to the client.

4.2 External Scoping of Discard-based Search

Many anticipated use cases of Manna involve rich metadata that annotates the raw data to be searched by content. The use of prebuilt indices on this metadata enables efficient selection of a smaller and more relevant subset of raw data to search. This can greatly improve search experience and result relevance. Figure 2 shows how scoping is implemented. The scoping mechanism recognizes that some searches may span administrative boundaries. Each administrative unit (with full autonomy over access control, storage

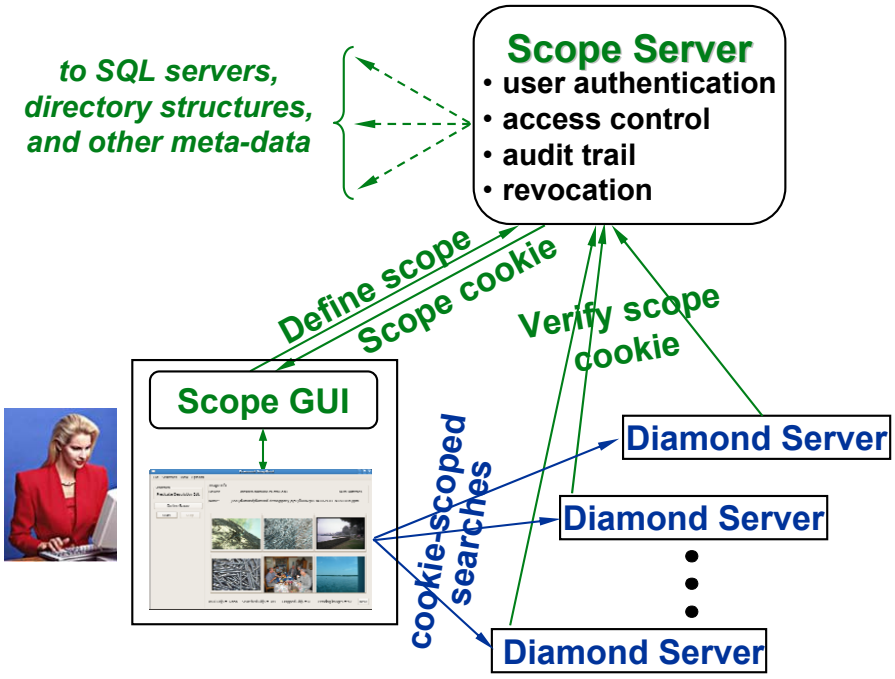


Figure 2: Scoping a discard-based Search

management, auditing, and other system management policies) is represented as a *realm* in this design. Realms can make external business and security arrangements to selectively trust other realms.

Each realm has a single logical *scope server,* that may be physically replicated for availability or load-balancing using well-known techniques. A user must authenticate to the scope server in her realm at the start of a search session. For each scope definition, a scope server issues an encrypted token called a *scope cookie* that is essentially a capability for the subset of objects in this realm that are within scope. The fully-qualified DNS hostnames of the Diamond servers that store these objects is visible in the clear in an unencrypted part of the scope cookie. This lets the client know which Diamond servers to contact for a discard-based search. However, the list of relevant objects on those Diamond servers is not present in any part of the scope cookie. That list (which may be quite large, if many objects are involved) is returned directly to a Diamond server after it validates the scope cookie. Figure 2 illustrates this flow of information. Scope cookies have finite lifetimes determined by the scope server, and are signed using X.509 infrastructure. For a multi-realm search, there is a scope cookie issued by the scope server of each realm that is involved. In other words, it is a federated search in which the issuing and interpretation of each realm's scope cookies occur solely within that realm. A client is only a conduit for passing a foreign realm's scope cookie between that realm's scope server and its Diamond servers.

A user generates a metadata query via a Web interface, labeled "Scope GUI" in Figure 2. The syntax and interpretation of this query may vary, depending on the specific metadata source that is involved. The scope cookie that is returned is passed to the relevant domain-specific application on the client, and then to the Diamond servers. When a user changes scope, new connections are established to relevant Diamond servers.

4.3 Complete Manna Architecture

Figure 3 shows how the subarchitectures of Figures 1, and 2 are combined with OpenDiamond to yield the complete Manna architecture. This figure is best understood by following the message flows that occur

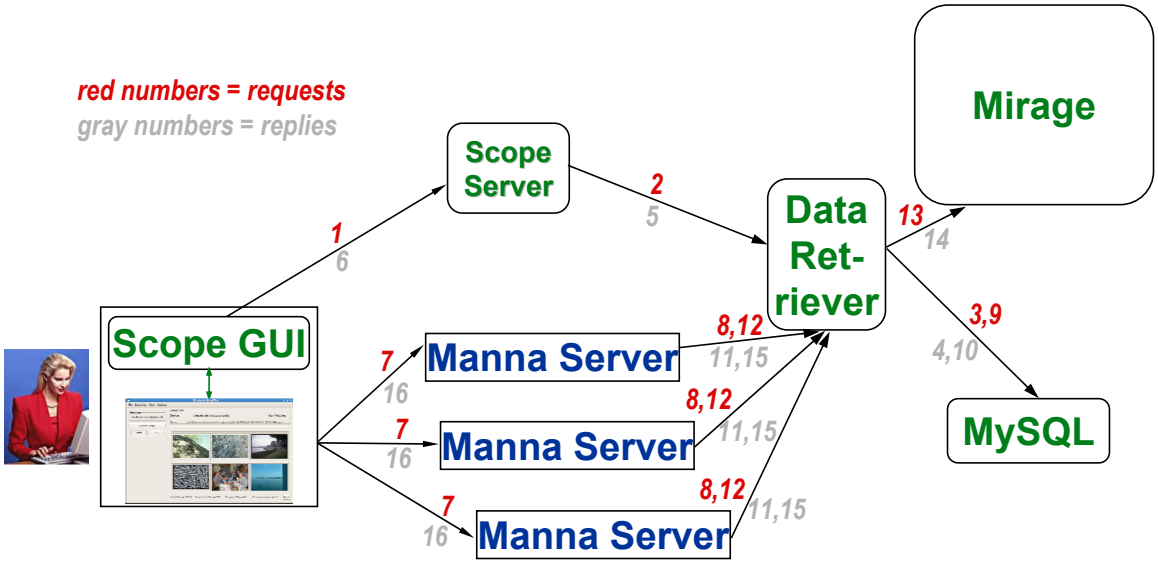


Figure 3: Manna Architecture

during a search. Using a Web browser, a user interacts with the scope server and defines the scope of a Manna search (1). The scope definition is encoded as a query component in an URI [2] and may include parcel attributes (such as parcel name, owner, committer, description, and creation time), as well as file attributes (such as pathname, mode, owner, group, and file size). The scope server constructs a scope cookie and presents it (2) to the scope method of the data retriever. The data retriever transforms the scope request into a SQL query and contacts MySQL (3) to identify the list of files on Mirage that are within scope. For example, the query string "`?path=%2a.jpg&path=%2a.gif`" is transformed into the SQL clause "`WHERE (file.path LIKE '%.jpg') OR (file.path LIKE '%.gif')`." Access control based on MySQL fields can be enforced at this point.

The MySQL result (4) is interpreted by the data retriever, which then validates the scope cookie and returns it via the scope server (5) to the client (6). Note that the cookie does not explicitly identify the list of files within scope. Rather, it only contains the information needed by the data retriever to regenerate this list in the future.

At the start of a search, the application sends the scope cookie to each server (7) that is involved in the search. Each server contacts the data retriever with its cookie (8), and gets back the list of Mirage files on which it is to perform discard-based search (9, 10, 11). Note that the distribution of work across servers happens implicitly at this point. Each server iterates through its list of Mirage files. For each file, it obtains the contents from Mirage (12,13,14,15) and performs early discard on it. That is, the file is examined by each plug-in (representing a constraint) the user has configured through the client GUI and if it is accepted by each plug-in, the object is returned to the user and not discarded. If the file is not discarded, a low-fidelity version of the file (i.e., a "thumbnail") is returned to the client (16). If necessary, the full-fidelity version of the file can be obtained on demand by the user.

5 Client Applications

To validate our prototype implementation of Manna, we have built three applications that are relevant to the use cases mentioned in Section 1: searching images to find examples that match a description, searching source code to find fragments that are similar to an example code fragment, and searching files for vulnerabilities or viruses. We describe these applications below in Sections 5.1, 5.2, and 5.3. These applications

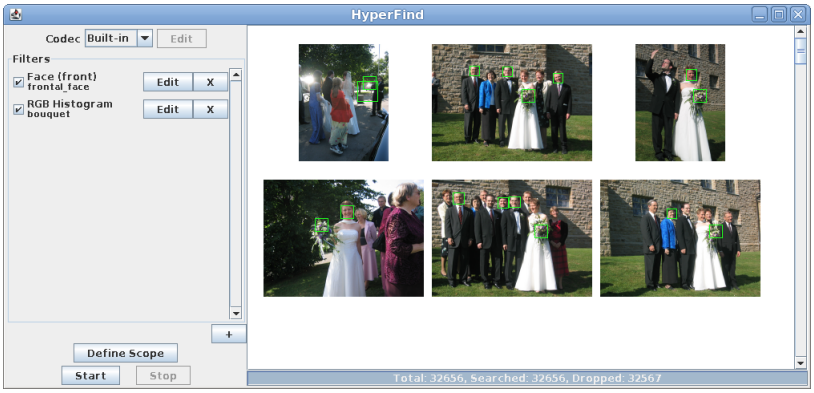


Figure 4: Example Search with HyperFind

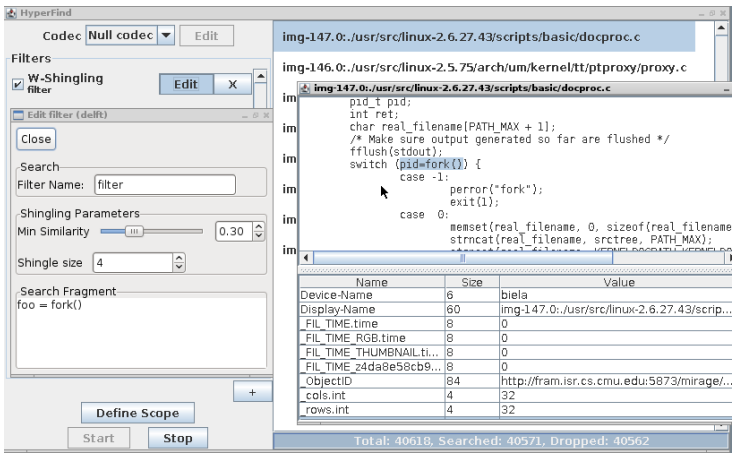
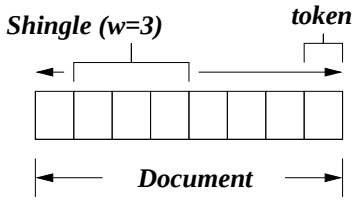


Figure 5: Example Search with ShingleFind

embody no knowledge of where the data is located.

5.1 Image Search with HyperFind

HyperFind, shown in Figure 4, enables users to interactively search large collections of unlabeled images. It could be useful in applications such as the graphic designers example mentioned in Section 1. HyperFind provides a GUI for users to create searches by combining simple plug-ins that scan images for patches containing particular color distributions, shapes, or visual textures. HyperFind supports plug-ins created using ImageJ, a Java tool that is widely used for image processing [4]. An investigator can create an ImageJ macro on a small sample of images, and then use that macro as a plug-in in HyperFind to search a large collection of images. A copy of ImageJ runs on each server to handle the processing of these plug-ins, and is invoked at appropriate points during search execution. A similar approach has been used to integrate the widely-used MATLAB tool. HyperFind also supports use of plug-ins that are automatically generated through machine learning techniques, specifically those based on the Semantic Texton Forest approach described recently by Shotton et al. [20]. HyperFind, the most mature client presented here, is implemented in 2648 lines of Java code built on top of the Manna client API. Since HyperFind uses many different plug-ins, their details are omitted for brevity.



Structure of a document (length B) and shingle (w contiguous tokens). Here, $w = 3$.

Figure 6: Shingling Concept

5.2 Source Code Search with ShingleFind

ShingleFind, shown in Figure 5, finds files whose content matches an example text using the technique of *w-shingling* [3] which is used to efficiently compute the resemblance of two documents. The shingling process involves constructing the set of all sequences of w consecutive tokens (shingles) for each document as shown in Figure 6. We then calculate the number of common values between these sets of shingles divided by the sum of all unique values, which gives a measure of resemblance over the range $[0,1]$. Instead of comparing token sequences, shingles are represented by their Rabin fingerprint. Rabin fingerprints have the property that given a fingerprint at offset n, we can efficiently calculate the fingerprint at offset $n+1$.

In our implementation, we are not comparing complete documents, but searching for matching fragments in the set of documents returned by scoping. To identify these fragments we search each document using a sliding window which contains the same number of tokens as the example. As we move the window one token at a time, we add one new shingle and remove a shingle from the set and adjust the union and intersection with the example set. We remember the offset that resulted in the best match and report it as the similarity value of the whole file.

The user can specify several parameters such as the shingle size, the minimum threshold value for similarity and how the input should be tokenized. A trivial tokenization treats each byte as a single token. A more useful tokenizer for source code would collapse sequences of whitespace into a single token and passes all other characters as is. Another option is a tokenizer that returns UTF-8 codepoints, and we intend to add the ability to add language-specific tokenizers/parsers that can understand the language grammar and can discover similarities in code structure even when individual functions or variables use different naming. ShingleFind is implemented in a 214 line C plug-in using the extensible Manna plug-in API, and a 240 line C++ client built on top of the Manna client API.

5.3 Virus and Vulnerability Search with ClamFind

ClamFind, shown in Figure 7, searches for viruses and potential vulnerabilities in files. Our approach is similar to the approach discussed by Wei et al. [23]; however, we provide a client GUI and all of the features of the Manna architecture. When applied to parcels through Manna, this provides a mechanism for system administrators to rapidly identify vulnerable parcels which may be in use by clients. ClamFind uses the ClamAV [6] engine for its scanning backend which provides the flexibility of using officially provided malware and vulnerability databases, as well as user provided custom signature databases. Systems administrators using the freely provided *sigtool* [6] can rapidly create signatures based on existing files and scan for them using ClamFind. This enables system administrators to search for unpatched systems, as in the example from the introduction.

The scan results shown in Figure 7 present an example of finding a potential vulnerability using an official signature database. The potential vulnerability identified in this case, *Suspect.PDF.ObfuscatedJS-1*, is a false positive residing in a real installation of Adobe Acrobat within a parcel loaded into our experimental dataset which is described in Section 6. ClamFind returns the path of the affected file, an identifying parcel name, and the name of the matched signature. ClamFind is implemented in a 119 line C code plug-in using the extensible Manna plug-in API, and a 186 line Java client built on top of the Manna client API.

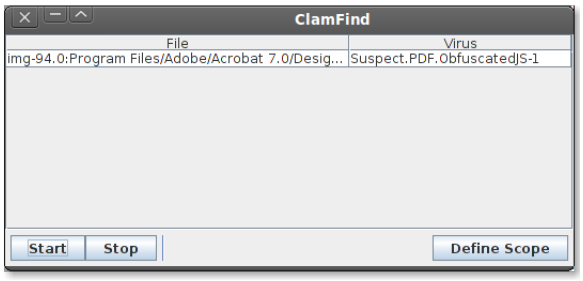


Figure 7: ClamFind Scan Results

Total parcels	88
Total files	$2.9x10^6$
Raw disk usage	306 GB
Dedup disk usage	89 GB

Figure 8: Statistics for constructed Mirage store at CMU.

6 Evaluation

In evaluating Manna we seek to answer three critical questions. What is the performance overhead of searching inside VM parcels versus searching local files? How scalable is the Manna architecture? What is an upper bound on the performance we can expect from Manna when considering different storage backends? The first question is answered through a trace-based approach using real searches collected from a user of Manna. The second question is answered through virus scanning, an embarrasingly parallel problem, and scaling the number of Manna servers taking part in the scan. The third and final question is answered through a series of microbenchmarks that investigate different storage backends, as well as different methods of accessing objects—for example, over a network versus local storage.

6.1 Performance Impact

For the first set of trace-based experiments, our experimental setup mirrored the architecture shown in Figure 3. The Manna servers were Dell Vostro 220's, each with an Intel Core 2 Duo 3 GHz processor, 3 GB of memory, and a 7200 RPM SATA drive. A single physical machine hosted the Mirage server, the MySQL database and the data retriever. This machine had two Quad Core Intel Xeon 2.66 GHz processors, 32 GB of memory, and a fibre channel RAID5 with 12 10K RPM drives. The client was a 2.66 GHz Intel Core 2 Duo Apple iMac with 4 GB of memory. All network connections were Gigabit Ethernet.

We populated the Mirage repository with 65 Windows XP parcels obtained from the VCL cloud at North Carolina State University. In addition, we synthesized and imported 10 parcels with Linux kernel source trees, and a further 13 parcels that together contained a Flickr dataset of 740,534 images. Figure 8 summarizes the contents of the Mirage repository. There were a total of 2.9 million files spread across 88 parcels. Deduplication was highly effective: only 89 GB of disk space was required to store 306 GB worth of files.

HyperFind was used to capture and replay the three traces summarized in Table 1. For each trace, the user was given a specific task and a soft time limit of 20 minutes. For Trace 1, the user was given a verbal description, "Find 10 images of beaches." Figure 9a shows a typical example of an image that the user found. For Trace 2, the user was presented Figure 9b, containing a crocodile. His task was to find images of similar composition, containing an animal, a tree and a wall. For Trace 3, the user was given Figure 9c. His task was to find other dog images.

Trace	Trace Task	Sample	Searched	Discarded	Viewed
1	Starting with no given image, find 10 images of beaches.	Fig. 9a	12140	11395	681
2	Starting with a target image of a crocodile outdoors with a tree, find at least one image with similar features (animal, outdoors, tree).	Fig. 9b	20427	20418	7
3	Starting with a target image of a dog, find 3 other images containing a dog.	Fig. 9c	20933	20365	560

Table 1: Trace Tasks and User Results

(a) Trace 1 (b) Trace 2 (c) Trace 3

Figure 9: Representative Images from Traces

	Trace 1		Trace 2		Trace 3	
Servers	Local Disk	Inside VM	Local Disk	Inside VM	Local Disk	Inside VM
1	988 (3.74)	948 (50.92)	1284 (2.49)	1325 (46.23)	1265 (2.87)	1274 (1.89)
2	549 (0.94)	549 (4.19)	678 (0.94)	680 (1.89)	743 (0.47)	743 (1.25)
4	334 (0.47)	334 (0.94)	375 (0.47)	376 (0.47)	480 (0.47)	483 (1.25)
6	260 (0.47)	261 (0.47)	275 (0.00)	274 (0.94)	396 (0.47)	393 (2.83)
8	225 (0.00)	231 (0.47)	226 (0.47)	225 (0.47)	355 (0.00)	356 (0.94)

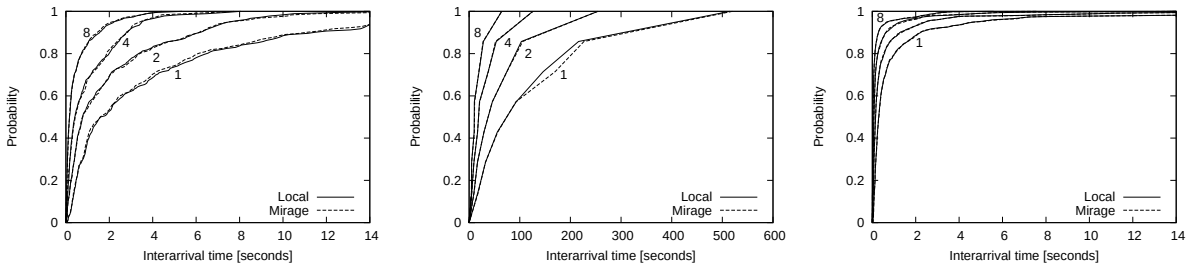
Each data point above is the mean of three runs. Standard deviations are in parentheses.

Table 2: Task Completion Time (Seconds)

	Trace 1		Trace 2		Trace 3	
Servers	Local Disk	Inside VM	Local Disk	Inside VM	Local Disk	Inside VM
1	11.8 (0.21)	12.4 (0.55)	16.9 (0.03)	16.4 (0.63)	25.5 (0.03)	25.0 (0.03)
2	24.2 (0.19)	24.1 (0.23)	33.8 (0.08)	33.7 (0.10)	48.8 (0.45)	48.4 (0.30)
4	46.6 (0.33)	46.2 (0.33)	67.5 (0.10)	67.4 (0.04)	89.9 (0.26)	89.0 (0.23)
6	66.4 (0.60)	66.4 (0.15)	100.5 (0.13)	100.9 (0.40)	119.6 (1.21)	121.4 (0.66)
8	84.1 (0.20)	76.3 (0.89)	132.6 (0.22)	133.3 (0.28)	142.7 (0.79)	143.0 (1.04)

Each data point above is the mean of three runs. Standard deviations are in parentheses.

Table 3: Discard Rate During Trace Replay (Objects per Second)



(a) Trace 1 (b) Trace 2 (c) Trace 3

The numbers on curves indicate number of Manna servers

Figure 10: CDF of Result Interarrival Times

Total parcels	1760
Total files	$1.7x10^8$
Raw disk usage	10.3 TB
Dedup disk usage	1.3 TB

Figure 11: Statistics for constructed Mirage store at IBM.

For each trace, Table 2 presents the time for task completion. For all traces, as the number of Manna servers increases, the task completion time decreases. This is because more computational resources are being devoted to the search. For each trace and number of Manna servers, there is at most a small increase in the task completion times on Mirage, relative to the local case. The worst case difference (Trace 2 at 1 Manna server) is about 3%. In most cases, the times are identical within bounds of experimental error. Table 3 presents the aggregate discard rate across Manna servers. Once again, there is little difference in the observed values between the local and Mirage case. The worst case difference (Trace 1 at 8 Manna servers) is about 9%. In most cases, the difference is much smaller. Figure 10 shows the cumulative distribution function of observed result interarrival times. For all traces and number of Manna servers, the curves for the local and Mirage cases are extremely close. This indicates that interactive user experience is virtually indistinguishable between the two cases. Taken together, the results presented in Table 2, Table 3, and Figure 10 confirm that the performance overhead of the Manna architecture is minimal. A user can hardly tell whether he is searching local or cloud data.

6.2 Scalability

The second set of experiments were conducted on a snapshot of 1760 images from RC2 [1] using the same architecture as shown in Figure 3. Figure 11 summarizes the contents of the snapshotted Mirage store. The experiments where themselves conducted inside RC2. The images were stored on a 2 TB volume that supports a read bandwidth of 1 GB/s. This volume was attached over the network to the file server, which was a "large" RC2 instance with 4 2.9 GHz virtual cpus and 6 GB of virtual RAM. A second 2 TB network-attached volume held the MySQL database. Each compute node was a "small" RC2 instance with 1 2.9 GHz virtual cpu and 1 GB of virtual RAM; in our experiments, the number of compute nodes was varied from 1 to 8. A 1 GB/s network interface connected the file server to the compute nodes and to the storage. The client machine was connected over a wide-area network to the compute nodes; this machine experienced minimal CPU and network load. All RC2 instances ran RHEL 5.5.

ClamFind was used to seach for a vulnerability within the RC2 cloud. In our evaluation, we searched for parcels containing an unpatched and vulnerable RHEL 5.3 kernel. We found no images that contained the kernel vulnerability, although we did not run complete searches due to time constraints.

Wei et al. [23], as mentioned in Section 5.3, introduced the notion of virus scanning with Mirage to greatly reduce the time needed to scan many VM images by examining deduplicated files versus examining

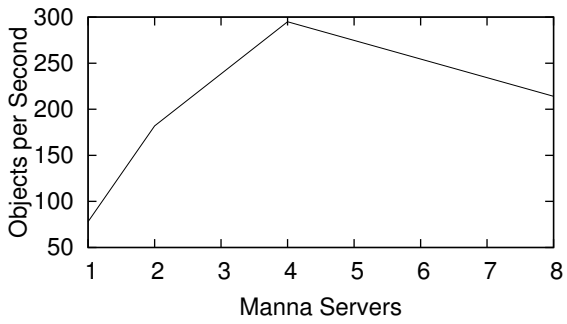


Figure 12: Virus scanning the RC2 cloud.

all files in all VM images. We produce results only for the deduplicated case, because we want to show how Manna scales.

To explore how well ClamFind scales within RC2, we ran ClamFind several times for 15 minutes each time, varying the number of compute nodes, and, for each run, recorded the average number of items scanned per second. The file server and all compute nodes were rebooted immediately before each run. Note that a sequential phase constructs the list of in-scope items. This phase, which took approximately one hour to complete, was run once ahead of time and the result was used for all runs.

Figure 12 shows the results. The best measured throughput, 295 items per second, was achieved with 4 compute nodes. With 8 compute nodes, performance degrades to 214 requests per second. Interestingly, with 8 nodes, the file server served only 22 MB/s to the compute nodes, which is well short of the 110 MB/s bandwidth achievable when sending zeroes to the same nodes. One conjecture is that this is caused, at least in part, by contention between the virtual network interface and the network-attached volumes for the same physical network. ClamFind uses both heavily and, in a microbenchmark, virtual network bandwidth falls by a third when a network-attached volume is under a high read load.

6.3 Quantifying the Impact of Storage Backend

The third set of experiments were conducted using two datasets: one consisted of 81309 JPEG images with an average filesize of 75 KB for a total of 5.5 GB of data, and the other was the VM dataset from the first set of experiments located within Mirage characterized in Figure 8. The dataset of JPEG images was used to benchmark retrieving objects from a HDD (1.5 TB Seagate Barracuda disk), SSD (Intel X25-M), and RAID1 (two 1.5 TB Seagate Barracuda disks). The test server for the JPEG dataset has an Intel Core 2 Duo 3 GHz and the test data was copied to an ext3 filesystem mounted with noatime. The VM dataset was used to benchmark retrieving objects from within Mirage over a network. The server backend for the VM dataset is the same as that in the first set of experiments. There are four types of microbenchmarks: raw read random, raw read linear, null plug-in, and `rgbimg` plug-in. The raw read microbenchmarks represent the full I/O bandwidth of the underlying storage backend when accessing data eit her randomly or linearly. The null plug-in represents a pass-through plug-in that performs no computation and presents the minimal cost of the Manna architecture. The `rgbimg` plug-in represents a plug-in with computation and exhibits the fundamental tradeoff between computation and I/O bandwidth.

Figure 13 shows different microbenchmarks performed on the different storage backends. The benchmarks have been chosen to highlight various bottlenecks that are typically encountered. In normal usage, Manna takes advantage of caching to mitigate I/O bottlenecks independent of the storage backend, but all caches were flushed for these experiments. Optimization of the storage backend could occur in two places: in the data retriever software layer, or in the hardware configuration.

A first-cut design for storage backend would not consider any software layer optimizations and use the most prevalent storage technology—HDD technology. The limitations of HDD technology are primarily

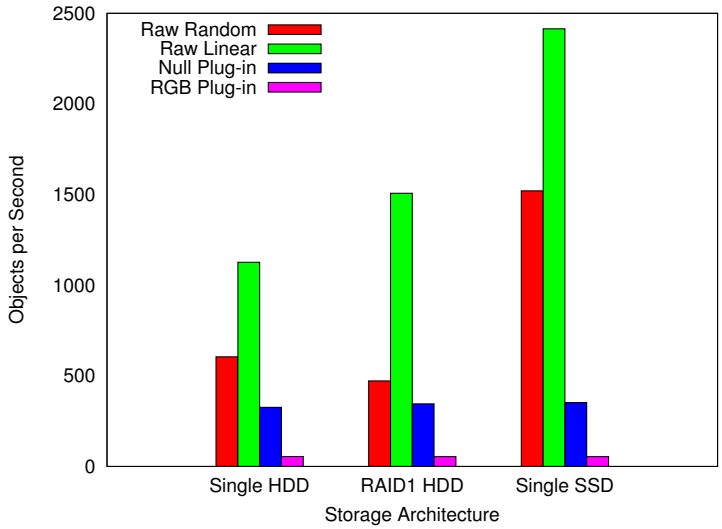


Figure 13: Microbenchmarks comparing several storage backends.

seek time and read rate. We designed our microbenchmarks to specifically measure the seek time cost. The first microbenchmark, raw read random, represents this first-cut scenario with no optimizations. There are no software optimizations—object access is random—and no hardware optimizations. Single HDD raw random read, exhibits the seek cost for accessing each object and is the slowest of the storage backends considered as shown in Figure 13.

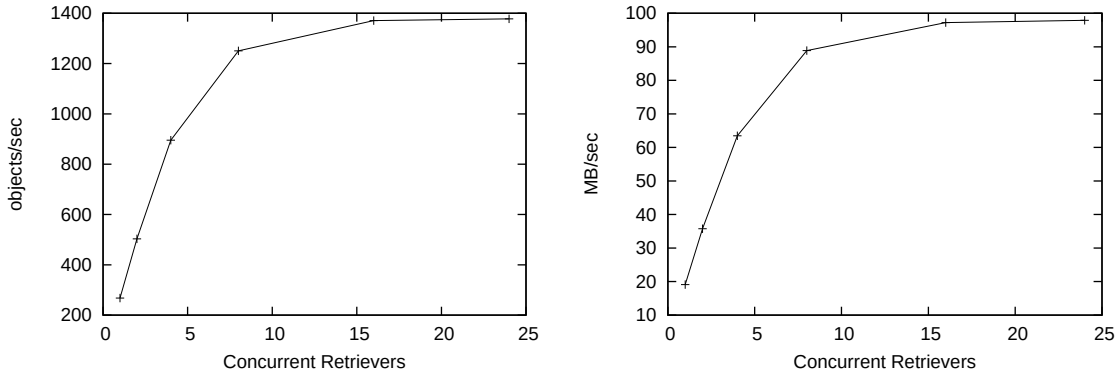
An improved design, without changing the hardware, involves a simple preprocessing optimization in the data retriever. Accessing objects linearly, as opposed to randomly, amortizes an initial seek time cost across all of the objects. For HDD technology this provides a boost in the objects per second that can be accessed—evident through the HDD raw linear read shown in Figure 13. This microbenchmark represents the best performance from an HDD using only software optimizations by taking advantage of readahead for consecutive objects which often have adjacent inode and file data allocations.

Unsatisfied with optimizations solely in the data retriever, we now turn to different hardware configurations. The simplest change is to continue using HDD technology, but using a RAID1 to reduce seek time penalties when accessing objects. Figure 13 shows a clear win for RAID1 in the raw linear read case, where the read rate from the RAID1 in aggregate is greater than the read rate from a single HDD. We did not notice an increase in speed for the raw random read case, a surprising find, but we conjecture this is due to inefficiencies in software implemented RAID1.

The final design is a hardware optimization that swaps HDD technology for newer SSD technology. SSD's do not suffer the penalty of seek time, and benefit from a faster read rate than HDD's. In both benchmarks, designed to show the cost of seek time, SSD's outperform HDD technology even when in a RAID1 configuration. We found that accessing objects linearly as in the optimized HDD case also increased the performance of SSD technology as the better locality of reference improves cache usage.

We must also consider the computation costs introduced by the Manna architecture because we must balance the speed of the storage backend with the rate of processing objects. Manna introduces computation costs in two places: the content addressable result cache, and the plug-ins specified by the user. The content addressable result cache is a part of the Manna architecture and represents a fixed cost. User plug-ins can vary highly in computational cost—from minimal identity functions to complex face recognition algorithms.

We identified the cost associated with the content addressable result cache by using a microbenchmark based on a null plug-in. The result cache uses a content hash of an object as the key to find previous plug-in execution results which may allow the search to discard the object without performing further execution or loading the object from the Mirage store. This shares the cost of a plug-in across future searches. The



(a) Object retrieval rate

(b) Network usage

Figure 14: Mirage data retriever performance.

dominating cost is hashing the object. Hashing is a computation-bound operation which, as reflected in Figure 13, we do not see any change depending on the storage backend.

We also used a simple JPEG decompression plug-in to show what happens when plug-ins with computation are added to the search on top of the result cache. We again see a drop in the objects processed per second, and no difference across the storage backends. We are computation-bound with the user plug-in and result cache. However, in a typical interactive Manna search we may find hits in the result cache and achieve objects processed per second rates equivalent to or even exceeding the null plug-in performance rates.

Finally, we performed a set of microbenchmarks on our existing prototype implementation as discuessed in Section 6.1. This is different than the microbenchmarks presented in Figure 13 because we go through every piece of the Manna architecture and traverse a network. In this case the Manna server retrieves objects from a Mirage server using HTTP requests sent to the data retriever. To identify the bottlenecks of this data retriever setup we benchmarked its performance using Siege [21] from one of the Manna servers. The results are shown in Figure 14. We observe a peak return rate of 1378 objects/second at which point we were transferring an average of 98 MB/second across the Gigabit ethernet—essentially saturating the network bandwidth.

We conclude from our microbenchmarks that SSD technology provides the quickest object access read rates, and that the Manna architecture is computation bound with cold caches and user provided plug-ins. However, with caching the performance of Manna can exceed the raw processed objects rates of the storage backend because the objects do not need to be retrieved when search results are cached.

7 Conclusion

We explored efficient search of cloud-based parcels to support administration of large collections of parcels. We identified the challenges with searching parcels and presented Manna, a plug-in architecture that addresses these challenges by combining three technologies: index-based search via standard databases and existing data structures, deduplication of parcel data via Mirage, and non-indexed search via the OpenDiamond Platform. We described three applications that use the Manna architecture, HyperFind for searching parcel-based images, ShingleFind for source code search, and ClamFind for virus and vulnerability search. We evaluated the performance of our prototype Manna implementation and showed that the prototype can perform searches with as little as 3% overhead over search on local (non parcel-based) storage, while delivering similar interactive experience. Finally, we showed that Manna scales well with the size of parcel collections, allowing administrators to simply add additional Manna servers as their parcel collections grow.

Acknowledgements

Vas Bala of IBM Research suggested the idea of applying discard-based search to VM images in Mirage. We wish to thank him and the rest of the Mirage team, including Glenn Ammons and Xiaolan Zhang, for the research collaboration that made this work possible. We gratefully acknowledge Peng Ning and Xianqing Yu of North Carolina State University for sharing their collection of VM images with us. We thank Richard W.M. Jones for his timely release of patches for building libguestfs on Ubuntu. We acknowledge the contributions of Benjamin Gilbert in helping us improve the presentation of this paper.

OpenDiamond is a trademark of Carnegie Mellon University.

References

[1] AMMONS, G., BALA, V., BERGER, S., DA SILVA, D. M., DORAN, J., FRANCO, F., KARVE, A., LEE, H., LINDEMAN, J. A., MOHINDRA, A., OESTERLIN, B., PACIFICI, G., PENDARAKIS, D., REIMER, D., RYU, K. D., SABATH, M., AND ZHANG, X. RC2: A living lab for cloud computing. IBM Research Report RC24947, IBM, 2010.

[2] BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. RFC3986 - Uniform Resource Identifier (URI): Generic Syntax. `http://tools.ietf.org/html/rfc3986`.

[3] BRODER, A., GLASSMAN, S., MANASSE, M., AND ZWEIG, G. Syntactic clustering of the web. In *Proceedings of the 6th International WWW Conference* (1997).

[4] BURGER, W., AND BURGE, M. J. *Digital Image Processing*. Springer, 2008.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation* (2006), vol. 7, pp. 205–218.

[6] CLAMAV. Clam AntiVirus. `http://www.clamav.net/`.

[7] CROFT, B., METZLER, D., AND STROHMAN, T. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2009.

[8] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G.R., RIEDEL, E., AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, April 2004).

[9] KOZUCH, M., SATYANARAYANAN, M. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications* (Callicoon, NY, June 2002).

[10] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[11] LIGUORI, A., AND HENSBERGEN, E. V. Experiences with content addressable storage and virtual disks. In *In Proceedings of the Workshop on I/O Virtualization (WIOV 08* (2008).

[12] MINKA, T., PICARD, R. Interactive Learning Using a Society of Models. *Pattern Recognition 30* (1997).

[13] NATH, P., KOZUCH, M.A., O'HALLARON, D.R., HARKES, J., SATYANARAYANAN, M., TOLIA, N., TOUPS, M. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-Scale Systems Based on Virtual Machines. In *Proceedings of the USENIX Technical Conference* (Boston, MA, June 2006).

[14] OPENDIAMOND. Interactive Search of Non-Indexed Data. `http://diamond.cs.cmu.edu`.

[15] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (Monterey, CA, January 2002).

[16] REIMER, D., THOMAS, A., AMMONS, G., MUMMERT, T., ALPERN, B., AND BALA, V. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *VEE'08: Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Seattle, WA, March 2008).

[17] SATYANARAYANAN, M., GILBERT, B., TOUPS, M., TOLIA, N., SURIE, A., O'HALLARON, D. R., WOLBACH, A., HARKES, J., PERRIG, A., FARBER, D. J., KOZUCH, M. A., HELFRICH, C. J., NATH, P., AND LAGAR-CAVILLA, H. A. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing 11*, 2 (2007).

[18] SATYANARAYANAN, M., RICHTER, W., AMMONS, G., HARKES, J., AND GOODE, A. The Case for Content Search of VM Clouds. In *CloudApp 2010: The First IEEE International Workshop on Emerging Applications for Cloud Computing* (Seuol, South Korea, 2010).

[19] SATYANARAYANAN, M., SUKTHANKAR, R., MUMMERT, L., GOODE, A., HARKES, J., AND SCHLOSSER, S. The unique strengths and storage access characteristics of discard-based search. *Journal of Internet Services and Applications 1*, 1 (2010).

[20] SHOTTON, J., JOHNSON, M., AND CIPOLLA, R. Semantic Texton Forests for Image Categorization and Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Anchorage, AK, June 2008).

[21] SIEGE. HTTP regression testing and benchmarking utility. `http://www.joedog.org/JoeDog/Siege`.

[22] VOUK, M., RINDOS, A., AVERITT, S., BASS, J., BUGAEV, M., PEELER, A., SCHAFFER, H., SILLS, E., STEIN, S., THOMPSON, J., AND VALENZISI, M. Using VCL technology to implement distributed reconfigurable data centers and computational services for educational institutions. *IBM Journal of Research and Development 53*, 4 (September 2009).

[23] WEI, J., ZHANG, X., AMMONS, G., BALA, V., AND NING, P. Managing security of virtual machine images in a cloud environment. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security* (New York, NY, USA, 2009), ACM, pp. 91–96.

[24] YAO, A., YAO, F. A General Approach to D-Dimensional Geometric Queries. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (May 1985).