44

Push-Based Web Filtering Using PICS Profiles

by

David M. Shapiro

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

April 9, 1998

Author _____
Department of Electrical Engineering and Computer Science
April 9, 1998

Certified by _____
Dr. James S. Miller
Thesis Supervisor

Accepted by _____
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Students

Push-Based Web Filtering Using PICS Profiles

by

David M. Shapiro

Submitted to the
Department of Electrical Engineering and Computer Science

April 9, 1998

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis demonstrates that "channels" created by dynamically filtering Web content are a viable large-scale information delivery mechanism. Using the techniques developed here, three desktop computers can, in one day, alert every Internet user to information of interest from 10,000 different Internet hosts. Similarly, in one day, nine desktop computers could be used to inform 100,000 users of all information of interest to them on the entire World Wide Web.

The core of this project is a profile store that converts user profiles expressed as PICSRules into efficient, equivalent C programs. These programs use metadata as input (in the form of PICS labels) to describe the content of Web pages. A series of careful measurements shows that the system scales linearly in both the number of labels and the number of users (profiles).

Thesis Supervisor: Dr. James S. Miller
Title: Research Scientist, MIT Laboratory for Computer Science

Acknowledgements

First of all, I'd like to thank my thesis supervisor Dr. Jim Miller. He was always able to provide me with helpful ideas, both during my research and while I was writing my thesis document.

I'd also like to thank the staff of the World Wide Web Consortium, for providing me with a comfortable work environment not only during my thesis year, but for the two years of undergraduate work that I did before that. I'd especially like to thank W3C for providing me with a Research Assistantship for my thesis year.

I'd also like to thank everyone at W3C who put up with having me live in their office at one time or another: Dr. Jim Miller, Philip DesAutels, Joseph Reagle, Ralph Swick, Ora Lassila, and Marja-Riitta Koivunen.

Finally, I'd like to thank my family for providing me support during my final year at MIT. I would not have finished it without them.

Table of Contents

List of Figures and Tables

1 Introduction

Can dynamic channels (push technology) perform well enough to be used as an alternative to the existing Web (pull technology) browsing? This thesis presents a computational engine that is capable of providing channels of filtered Web content to a very large number of users. Experimental data show that this system is capable of filtering 10,000 Internet hosts for information to be distributed to every user of the Internet, using only three desktop computers for a single 24-hour period. Similarly, nine desktop computers could be used to provide 100,000 users with filtered information from the entire World Wide Web, in a single day of processing.

The central idea of this thesis is a profile store that converts user profiles expressed as PICSRules into efficient, equivalent C programs. These programs use metadata as input (in the form of PICS labels). From this metadata, the profile store determines which sources of information are relevant to particular users, according to their profiles. This information can then be distributed so a user receives only information that has matched his or her profile.

Today, search engines and catalogs are the primary way for users to deal with the onslaught of new information. Companies like Lycos, Yahoo, and Excite, developed products that provide users with the ability to search the Web. Products by these companies, and others, all attempt to index the entire Web in a searchable manner. All of these products have one common failing: users must actively visit these index sites to find what they want. While this is an improvement over flailing blindly through the Web, it can still be a time consuming task. Sorting out the useful information from the junk becomes a bigger and bigger task as these indices grow.

One solution to user's problems in locating useful information is to directly provide such information for the users rather than make them search for it. By using a profile for each user, it can be arranged so that each user receives information that is actually relevant. This is sometimes referred to as a channel. Channels are simply streams of information that are targeted for a particular user. Each user controls their channel by means of their profile.

This thesis demonstrates a channel system that uses industry standard filtering technology to provide users with content that matches a pre-defined profile in an efficient manner. Using the techniques described here, it is possible to build a scalable system that provides one channel per user, delivering information dynamically selected for that particular user based on an individual filtering profile.

1.1 Purpose

The goal of this project is to create a system which can deliver filtered content using a push-based model rather than the traditional pull-based paradigm of the Web. The system makes use of the PICS technology standards, created by the World Wide Web Consortium (W3C), as its filtering mechanism. This system demonstrates that label-based filtering via PICS is a reasonable alternative to content-based filtering in terms of performance and features.

1.2 Push vs. Pull

Push and pull are different models of information transmission. The World Wide Web is based on the pull concept. A user requests a page before it is sent to him or her. The request is the "pull" that signals a Web server to send a page to that particular client.

Push mechanisms behave differently. A push-based design has information sent to the user before the user has requested it. Many current push systems make use of user profiles in order to better determine what information a particular user actually wants to see. Other systems have users select a style of contents from a group of pre-determined data streams. A push-based system that consistently retrieves information that is not to the liking of its users would be aggravating and useless.

The system I designed takes the push concept and applies it to the Web in order to filter out unwanted content. The results of the filtering operation are supplied to the users to indicate that particular URLs have passed their filtering profiles. All of the filtering computation is done on the server according to the inputs that it receives. The profile store compares PICS labels to user profiles to determine which user profiles accept a particular URL. A complete description of the entire system is contained in Chapter 2.

1.3 PICS

There are many different filtering technologies in use on the Web today. The three most popular methods in use are list-based, content-based, and PICS-based filters. I have chosen to use PICS as my filtering mechanism. For more information about the other filtering methods, see Section 1.4.

PICS, the Platform for Internet Content Selection, is "An infrastructure for associating labels (metadata) with Internet content." PICS was initially developed by W3C for the purpose of child protection, but its open-ended design allows it to be used for a variety of other applications for which having labels would be helpful. For filtering purposes, PICS allows users to construct filters that look only at the labels for pages, rather than at the pages themselves.

There are three core datatypes associated with PICS. They are: rating service descriptions, labels, and profiles. Together they can be used to build powerful filtering tools. Each one is an important component of this system.

1.3.1 Rating Services

Rating services are machine readable descriptions of types of labels. In simple terms, a rating service file contains a list of categories that a label can rate. Each category contains information about what values are valid ratings and what the meanings associated with values are. A rating service is necessary in order to construct or read labels. Several popular rating services exist, including RSACi, SafeSurf , Net Shepherd

and SurfWatch. For more information about PICS rating services, see the rating service specification [PICS96a].

1.3.2 Labels

Labels are the actual indicators of the ratings associated with Web content. Labels correspond directly to a particular rating service. Knowledge of that rating service is necessary in order to comprehend the meaning of the label. Pages can be labeled in a variety of ways. Labels can be embedded in the headers of HTML pages, allowing Web page authors to provide ratings for their own work. Labels can be transmitted as a header in a RFC-822 datastream, providing a useful transport mechanism for authors to rate other documents of their own that are not actually in HTML. Finally, labels can be totally separate from a document, allowing for third-party rating of Web pages. A variety of label bureaus now exist that contain labels representing a particular organization's ratings. For more information about PICS labels, see the label specification [PICS96b].

1.3.3 Profiles

Profiles are the newest component of PICS. Using a new language known as PICSRules, users can create a document that precisely defines the behavior that they want their filter to have. Since PICSRules profiles are simply ASCII text, they may be freely exchanged among users. This allows users who are not comfortable with constructing their own profile to use one provided by a trusted source. PICSRules is a very new technology, having just received W3C Recommendation status in December 1997. For this reason, it is not yet in widespread public use. In the future, it is expected that users will be able to visit Web sites containing profiles, read English descriptions of what they do, and download the profile that matches their filtering preferences. For more information about PICSRules, see the PICSRules specification [PICS97].

1.4 Other Filters

As mentioned in Section 1.3, there are several other technologies that are in use today for the purpose of filtering Web content. A brief list of these follows, with some of the problems associated with them.

1.4.1 URL Lists

One way to filter out unwanted content is to construct a filter that looks only at URLs and matches them against a known list of offending URLs. This is commonly referred to as a "blacklist". Only URLs that are not on this list can be viewed by the user.

Another simple list based mechanism is to do the exact opposite. By maintaining an acceptable list of URLs (often called a "whitelist") and only allowing access to those

pages, a very restrictive filter can be created, while still allowing access to specific sources of information.

Both kinds of lists can be applied to filtering push-based content. Information that is being received along a channel can be filtering based on the source of that information. Every piece of information will have an Internet host as a source, and possibly a permanent URL as well. Comparing the URLs or even just the hostnames of the sources of the information against whitelists or blacklists can provide a coarse information filter. Users could receive specific kinds of information by placing those hosts on a whitelist. Users can block out other kinds of information by placing those hosts on a blacklist. This mechanism is not as flexible as either PICS or content-based filters.

1.4.2 String Matching

Another popular filtering mechanism is the content-based filter. By comparing the actual contents of pages against lists of strings, pages can be accepted or filtered based on their actual contents rather than just their URL. The benefit of this method is that it is easier to maintain than a list-based mechanism. Several difficult problems arise when using this method, however. First, creating the heuristics to determine if a page is acceptable or not can be difficult. There is also a significant amount of computation that needs to be done on every page that is accessed in order to see if it is to be accepted or blocked. This can cause noticeable slowdowns in the performance of the user's browsing. For this reason some systems use a database to store information about pages' contents rather than performing the same computations of every user's machine every time that want to view a particular page.

1.5 Related Work

There is a variety of existing work that has dealt with problems similar to those that this thesis addresses. Some are other Web filtering mechanisms, others involve similar data processing and distribution problems. The next few sections briefly describes several such existing projects. This is meant to be an overview of the area of research rather than a comprehensive listing of all related projects.

1.5.1 Query Processing for Information Distribution

One of the primary inspirations of this project is the work described in [BM94]. It addresses a similar problem, using user profiles to filter information. This project was a totally string-based system used to filter information from USENET news. It describes several stages of optimizations, ultimately leading to a performance level of 35,000 articles vs. 100,000 profiles in 8 hours. These numbers were taken into consideration while I was determining the initial performance goal of this thesis.

The design issues raised in [BM94] formed a starting point for related problems that I had to overcome in this thesis. The notion of using the profiles as data rather than queries was an important lesson from [BM94]. Furthermore, the idea that preprocessing could dramatically improve performance was introduced in [BM94], and eventually made it into the final prototype of my thesis design.

Although the details of the problem being addressed in [BM94] are rather different, it served as a good source of general ideas that I later adapted and specified for in my own research. Based on the experiments in [BM94], I created my initial design goal for processing of 10,000 labels against 100,000 profiles in 8 hours.

1.5.2 HTTP Distribution and Replication Protocol

In August of 1997, several companies jointly submitted a document for the HTTP Distribution and Replication Protocol [DRP97]. This document contains technical details for an extension to HTTP to allow for easier implementations of subscription-based push protocols.

The notion of using push to deliver subscriptions of Web content has become somewhat popular. Using technology that exists today, it is not difficult to create software that downloads new Web pages to users by using a subscription list and timestamps. Users subscribe to certain pages (or sites) and a program running on the server can communicate with a program running on the client to determine which pages have changed during a given time interval. The client side application can automatically download these pages and either store them or display them. This notion of push by subscription solves the problem of Web pages becoming out of date by resending them to the users that subscribe to them.

DRP is an extension to HTTP designed to provide a backward compatible add-on to HTTP that will make distribution of Web content more efficient. Under standard HTTP, different software vendors had to create their own solutions to the problem of maintaining up-to-date pages on a large number of client machines. DRP is an attempt to standardize this processes within the HTTP protocol. It remains to be seen if this will be a successful idea or not.

1.5.3 ANATAGONOMY

NEC Corporation's labs have created a push-based system known as ANATAGONOMY. This system, described in [KSK97], pushes news articles at users

based on user profiles. Unlike most user profile systems, this thesis included, their system does not force the user to make profile settings, or register for particular categories of information. The ANATAGONOMY system feature automatic personalization, meaning that it observes what documents a user requests, and builds a profile for that user by extracting keywords from those documents. Subsequent documents are scored against the existing profile in order to display to the user the most relevant documents available, based on the user's profile. In this manner, the system constantly modifies the user's profiles based on which documents the user reads. The user is given information in the order of calculated score. The goal is, over time, to have the system learn a user profile for each user that is good enough so that most high-scoring documents are truly important, and most documents with low scores are actually unimportant to the user. This system has been used in testing to deliver news stories from newspapers in Japan.

2 A Total System

The system I have designed consists of four components. They are a profile creation interface, a label fetching robot, a profile store that evaluates profiles and labels, and an end-user output generator. Together these four pieces form a complete system. First, users construct their profile and submit it to the profile store. The server on which the profile store resides does the rest of the work. It retrieves labels from the Web and evaluates them against all of the stored profiles. As a result, a list of acceptable URLs is generated for each profile. The output generator can then provide an individualized list of URLs to every user in the system. The list provided to each user contains only URLs that matched their profile.

2.1 Profile Creation

Users have several different options available for profile generation. Users do not need to construct their own profiles if they choose not to do so. Each profile is a document in the PICSRules format, and is therefore a freely transportable ASCII text file that can be sent to others. Users, if they so choose, can use a profile that is provided to them by some outside source. This source may be someone the user knows personally, or it may be a trusted collection of profiles. Users can transfer profiles electronically just like any other text file. Collections of user profiles are currently in the process of being created. When completed, a user will be able to go to the profile site with a Web browser, read English descriptions of the various profiles available, select the one they want to use, and download it to their machine. By acquiring a profile from an outside source, the user need not learn the exact details of how PICSRules works in order to use profiles constructed with it.

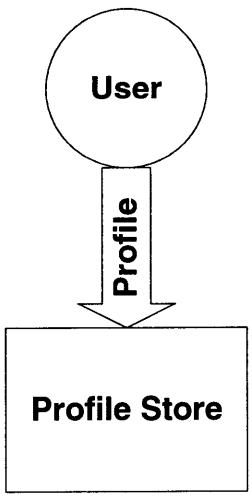

Figure 1 – Step 1: User Submits a Profile

Alternatively, users can use a profile construction program. I have created a prototype profile editor with a graphical user interface [PICSPC]. It is capable of reading PICS service description files and generating on-screen editors for each service. The user can select options for each category in each service and write the result out to a profile. Users also have the option to load previously created profiles and modify them, rather than start from nothing. With further work, a program such as this will allow users to create arbitrary profiles while presenting an intuitive interface. Although some understanding of PICSRules is necessary in order to operate a tool like this, the benefit is that the user is in complete control of their profile, without having to rely on another source to supply one.

As with any text-based data structure, power users are capable of creating their own, perfectly usable profiles, while using only a text editor. The PICS data formats are well defined, and users that are very familiar with them can create profiles faster by hand than by using a graphical interface. This method is not encouraged for users who do not fully understand PICS data formats.

2.2 Label Retrieval

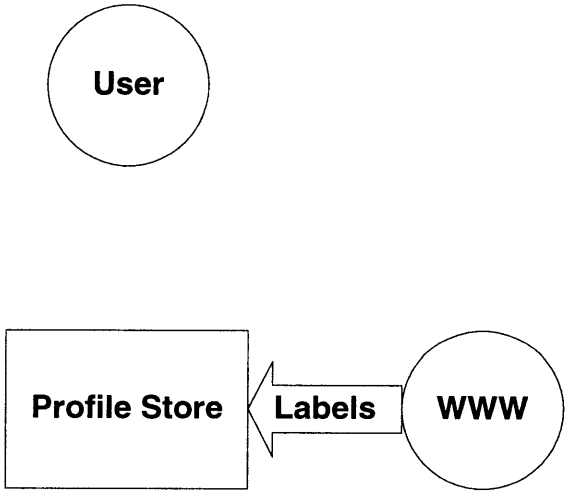

Figure 2 – Step 2: Labels Gathered from WWW

There are two different methods I considered for retrieving labels from the Web. This first way is to use a Web robot. Many different Web robots, often called spiders, currently exist that can roam the Web, usually for the purpose of collecting URLs. In my system I have experimented with an existing W3C Web spider, LabelGrabber [PICSLG], that captures and returns labels. The spider can retrieve labels by parsing the headers of HTML files looking for labels.

Instead of using a spider, it is also possible to gather labels by simply querying label bureaus. The benefit of this approach is that no Web-roaming spider is necessary. A label bureau can be queried repeatedly, possibly transferring its entire store of labels. An approach such as this is much easier to implement.

2.3 Profile Evaluation

The most important component of this system is the profile store. It collects information about all of the profiles and processes labels as they arrive as inputs. The processing of labels against the stored profiles can be set to occur during off-hours, when no users will be likely to be accessing the server. It has been carefully designed to achieve high performance during profile/label evaluation. The profile store is covered in full detail in Chapter 3.

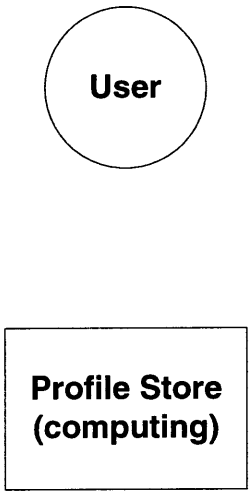

Figure 3 – Step 3: Profile Store Computes Acceptable URLs Overnight

2.4 Output Generation

The output generator takes the results from a profile store processing session and generates some form of output for the users. This output can take several forms. An HTML page can be created for each user, containing links to the pages whose labels matched that user's profile. Alternatively, each user could receive an email message containing the URLs which matched their profile. With further development, a system could be created that allows users to configure the manner in which they receive notification. This will allow customization of the output format (HTML, text, etc...), the frequency of updates (daily, weekly, etc...), and possibly other features.

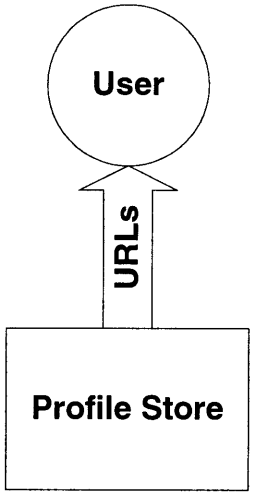

Figure 4 – Step 4: Profile Store Sends Acceptable URLs to User

2.5 Overall System Summary

From the user's perspective, this complete system asks for a profile once, and forever after provides lists of acceptable URLs on a regular basis. Due to the precise nature of PICS, no user will ever receive a URL that did not match the rules listed in their profile. Should a user become dissatisfied with the URLs that they received, they can change their profile to make it a better match for their needs.

The cornerstone of this system is the profile store. It must be capable of handling large numbers of both profiles and labels. It must also have realizable requirements of time and space. As you will see in Chapter 3, this was a difficult problem to solve, requiring a complex solution.

3 The Profile Store

The design of the profile store was the most challenging and most important piece of this research. It is the profile store which provides the computational muscle that allows for the processing of large numbers of profiles and labels within a reasonable amount of time.

3.1 Design Goals

One of the goals of my profile store design was to demonstrate that evaluating profiles against labels could be arranged in a manner that supported a large number of users within reasonable time and space constraints. My initial goal was based on the work in [BM94] (see Section 1.5.1). [BM94] demonstrated a system that could process 35,000 articles against 100,000 profiles in 8 hours. From this, I constructed the goal for my system: the ability to have over 10,000 labels run against 100,000 profiles in an 8 hour period. As you will see in this and later sections, reaching this goal required a design rather different than that of a traditional database.

A typical database would store the data, in this case the labels, and users would make queries on it (their profile selections). Instead of using this model, I inverted it, swapping the table data and the query information. The profile store contains representations of the user profiles, and the labels are run against it as the queries. This results in much greater performance because it reduces the amount of computation that needs to be done. This will be further explained in Sections 3.3 and 3.4.

In order to simplify the design process, I used only labels and profiles that made use of the RSACi rating service (see Appendix A for the actual RSACi service description). This choice does not limit the relevance of my experiments, since RSACi is one of the most popular PICS rating services in use today. Furthermore, rather than spend a lot of time gathering labels from the Web, I generated my own. Since the PICSRules standard was finalized while I was writing this thesis (long after I began work on this project), I had to create all of my own profiles, since none yet exist out on the Web. Further information about these details can be found in Section 4.1, "Experiment Goals".

In broad terms, the profile store is supposed to store the user profiles and use incoming batches of labels as queries. The results are handed over to the output generator.

3.2 Evolution of the Prototype

Designing a storage and query system that had adequate performance was not a simple task. My functioning prototype went through three major designs before I settled on the fourth and final design. A brief description of each of the early designs is included here.

The first design was written entirely in Java. It used hand-made parsers for the labels and profiles, and a hand-made table system based on the Java Hashtable class. This system was very fast for small data sets but it did not scale well. It consumed enormous amounts of memory.

My second design was also written entirely in Java. It was based on the first, but used a SQL server for data storage. Using JDBC, it generated SQL queries to access the profiles and labels from the SQL server. This implementation solved the problem of memory usage but its performance suffered. The huge number of SQL queries that were being evaluated resulted in long processing times for even small datasets.

My third design was a complete rewrite. I had just finished a project for W3C known as the PICS Standard Library. It included a fast set of parsers for profiles and labels as well as a new evaluator. I wrote a completely new system on top of these components. The result was a system that performed better than the previous Java implementation, and had reasonable memory requirements. Although it scaled linearly, the problem with this version was that it was not even close to reaching my original performance goal of 100,000 profiles vs. 10,000 labels in eight hours of processing.

My fourth and final design finally met the requirements I had set for time as well as requiring only reasonable amounts of space. As the data is Chapter 4 will show, the final version of my system was able to compare 100,000 profiles against 1.1 million labels in eight hours of processing time.

3.3 Final Design

My final design adds complexity to the code in exchange for speed. The complexity consists of a preprocessing stage in which both the labels and profiles are converted to formats better suited for rapid evaluation. As the remainder of this Chapter will show, this design eliminated all redundant calculations of label values against expressions within profiles, resulting in performance above and beyond the original design goals.

The label preprocessor is not very complex. For each label, it generates a text file starting with the URL that the label rates, followed by a string of ones and zeros. The string represents the results of running the label against all possible RSACi simple expressions. See the next Section "Making RSACi Run Quickly" for more information about this transformation.

The job of the profile preprocessor is to convert text versions of the PICS profiles into binaries that can quickly determine which labels match which filtering rules. I designed the preprocessor as a Java program that would output C source code. This C code could then be compiled into binary form as an executable file. This required the three steps that will be covered in detail in Section 3.5.

Profiles need only be preprocessed once, so I do not count this time as actual processing time. A set of preprocessed profiles can be run against different labels sets over and over without having to repeat preprocessing the profiles. Thus, the cost of preprocessing is amortized over many sets of labels, since the preprocessed profile set is used repeatedly. New batches of labels must be preprocessed each time, but the time for

preprocessing labels is negligible. For reference, I have included some actual preprocessing times in Chapter 4 with the experiments.

3.4 Making RSACi Run Quickly

For my experiments I used only the RSACi rating service. This service defines only four categories **{violence (v), sex (s), nudity (n), language (l)}**. Furthermore it defines only five values for each category, **{0, 1, 2, 3, 4}**. PICSRules supports the following relational operators: **{<, >, <=, >=, =}**. A PICSRules simple expression takes the form **{category relop constant}**. Example: **(rsaci.v < 2)**. These simple expressions are combined within a profile with boolean operators in order to produce complex user preferences. For an example of an entire user profile see Section 3.5.2. The result of this analysis is that the RSACi rating service can only produce 100 different simple expressions (4 categories x 5 values/category x 5 operators/value). In order to maximize the performance of the evaluation system, I designed the preprocessor to remove redundant computations.

The preprocessor evaluates each label against the 100 RSACi simple expressions exactly once during preprocessing, thus eliminating redundant calculations. These values are stored with the label so that during the actual evaluation stage of the system no comparisons of labels against simple expressions have to be made. Since all possible comparisons have been made during the preprocessing stage the evaluator needs only to look up the results of those computations. In this manner, the large number of computations required for label vs. profile evaluation during the actual processing stage have all been reduced to table lookup.

Although this design works quite well under the constraint of using only the RSACi rating service, it is not totally general. It could be adapted rather easily to use multiple rating systems, so long as they each had a finite number of values for each category. In fact, many of the popular rating services do fit this constraint. However, the PICS definition of a rating service, as given in [PICS96a], allows for rating services with non-labeled values. For these non-label-only rating systems, precomputing all possible simple expressions is not feasible. There are an infinite number of possible values regardless of whether the rating system uses integers only or all real values. An alternate system of preprocessing must be created in order to deal with non-labeled rating services. This is one of several areas outlined for future work in Chapter 5.

Example of Label Preprocessing

Original Label:

```
1  (PICS-1.1 "http://www.rsac.org/"
2    by "LabelMaker 2.0"
3    labels on "1997.11.19T09:57-0500"
4      for "http://www.foo.edu/"
5      r (v 3 s 0 l 2 )
6  )
```

Preprocessed Label:

```
1  http://www.foo.edu/
2  01001010010100100111110010
3  00111100101001010010010010
4  00000000000000000000000000
5  01001010010011111001010010
```

Line numbers have been included in both examples for legibility. They are not actually part of either the original or preprocessed label text. This label was constructed by a random-label generator. The label and its ratings do not correspond to any existing Internet host.

In the original label, the URL which is being rated is found on line 4, in the **for** clause. This URL appears (unchanged) in the preprocessed label on line 1. The original label places all of its rating information in its rating clause, indicated by "**r**" and located, in this example, on line 5. As explained in [PICS96b], the rating clause consists of a list of category names and values. The ordering of categories in the original label is not important. The ratings of the original label found on line 5 are parsed and evaluated by the preprocessor. The results of the evaluation are lines 2-5 of the preprocessed label.

To produce those lines, the preprocessor takes the ratings and evaluates all possible RSACi simple expressions against the values that appear in the actual label. Each evaluation returns either a zero or a one, depending on whether the expression being evaluated is false or true. There are one hundred result values produced for each label, twenty-five of them (5 values x 5 operators/value) corresponding to each of the four RSACi categories.

These values are written out to the preprocessed label in a fixed order. This order is predetermined by a conversion table. The order of expression in this table is shared with the profile preprocessor, allowing preprocessed profiles to later make use of the preprocessed labels. Details about the profile preprocessor can be found in Section 3.5.

In this example, line 2 corresponds to all RSACi simple expressions that require the category violence (**v**). In this label, **v** has a value of three, as seen in line 5 of the input label. Line 2 indicates the results of evaluating **v** as three in all possible RSACi simple expressions that use the category **v**.

Expression	Output (for v = 3)
rsaci.v < 0	0
rsaci.v > 0	1
rsaci.v = 0	0
rsaci.v <= 0	0
rsaci.v >= 0	1
rsaci.v < 1	0
...	...

Table 1 – Sample Conversion Table (Labels)

Table 1 demonstrates how these values are computed. All the RSACi simple expressions are arranged in the conversion table in a predetermined order. This label, with its **v** value of 3 is evaluated individually against each simple expression, producing a one or a zero for each expression. These are written out in the order in which they are produced. This process is repeated for each of the four categories (**v**, **s**, **n**, **l**) in the RSACi system.

Once a label has been preprocessed in this way it can be read by the profile store and used as a query. Preprocessed labels are read into an array by the C programs that represent the profiles. Since the profiles know exactly where to find their simple expressions in the array (the profile preprocessor uses the same order of expressions for its own conversion table), the actual processing of profiles and labels really is table lookup rather than further computation.

3.5 Profile Preprocessing

The profile preprocessor is the primary source of this system's speed. The profile preprocessor converts the ordinary PICSRules profiles into a more useful machine-readable format. This process requires three steps: Chunkification, Conversion and Compilation. It is the second step, Conversion, that contains the real intelligence of the preprocessor. It is the Conversion step that does the actual transformation from text to machine readable format. The output of the profile preprocessor is a single binary executable representing all of the profiles. This executable is the essence of the profile store: it reads the labels as inputs and determines which user profiles accept which labels.

3.5.1 Step 1 – Chunkification

The purpose of this step is to break up the profiles into more manageable chunks (pages). A command line argument allows the administrator of the system to control the size of each page. The purpose of this step is to prevent the machine from running out of memory while preprocessing. By operating on small groups of profiles at a time, there is no danger of running out of memory for the Java VM.

To accomplish chunkification, I implemented what is basically a virtual memory scheme for the input profiles. A single page of profiles is read into the program and then

the preprocessing takes place. In all of my experiments I used a page size of 200 profiles (the default). Once a page of profiles has been preprocessed those profiles are no longer needed internally by the preprocessor and they are replaced by reading in the next page of profiles. I did not do any experiments varying the page size, but when I briefly tried pages with more than 200 profiles I ran into memory and compilation problems during preprocessing. With very few profiles, performance suffered because of the cost of preparing each loop within the preprocessor.

3.5.2 Step 2 – Conversion

The conversion from PICSRules profiles to C source code is the key idea that provides the speed for the entire system. Rather than having an unwieldy mechanism that stores parsed profiles in some object form (as in earlier design prototypes) the use of C source code as an output format allows computations to be minimized. The amount of information required to store each profile as C code is quite small. The resulting executable is doing nothing more than performing evaluations. The evaluations themselves have been heavily optimized; every sub-expression possible is evaluated exactly once. Each profile then becomes a boolean combination of values, with each value determined by table lookup rather than by new computations. As soon as a label is known to be accepted or blocked by a particular profile, the rest of that profile is skipped, preventing needless computations.

For each page created in the first step, the preprocessor generates a single C source code file. Each profile within that page is turned into a boolean expression. A detailed example of this process is located at the end of this section. The C code created by the preprocessor is designed to expect the labels to be read from files when the program is run. This stage also creates a single Makefile for compiling and linking the resulting source files.

Profile Preprocessing Procedure

The profile preprocessor ignores all portions of the PICSRules profile except for the Policy clauses. The Policy clauses are read and translated, one by one, in the order that they appear in the original profile.

The system prototype I designed did not make use of the PICSRules AcceptByURL and RejectByURL Policy clauses that are used for making decisions based on URL (i.e. name) rather than content. Designing a computational engine for URL wildcard matching is a much simpler problem and well-understood. Adding this capability is straightforward and should not noticeably impact performance.

The remaining four types of Policy clauses are translated, while preserving the order in which they appear in the original profile. Translation requires the following steps for each profile.

1. Parse a Policy clause to determine the general form of the C statement to generate.
2. Convert each simple expression within the Policy clause into a C array reference, by using the conversion table.
3. Construct a C statement, according to the framework determined in the first step, using the expression translations created in the second step.
4. Repeat steps 1-3 for each Policy clause in this profile.
5. Once all of the Policies for a profile are translated, add an additional line to represent the default behavior of the profile (if a label does not get accepted or rejected by any of the existing Policy clauses, then that label is to be accepted).

Step 1 – Policy Clause Conversion:

The four types of Policy clauses translate into different types of C statements:

PICSRules	C
Policy (AcceptIf "...")	[else] if (...) result[currentlabel][usernumber] = 1;
Policy (RejectIf "...")	[else] if (...) result[currentlabel][usernumber] = 0;
Policy (AcceptUnless "...")	[else] if (!...) result[currentlabel][usernumber] = 1;
Policy (RejectUnless "...")	[else] if (!...) result[currentlabel][usernumber] = 0;

Table 2 – Policy Clause Conversion

The **...** represents the space where the simple expressions exist before and after translation. The **[else]** represents a portion of the statement that is not always present. It is explained under step 4. The **!** in the third and fourth conversions represents the C negation operator. The contents of the parentheses for those conversions are to be prepended with the negation operator, once they have been converted in the next step.

Within a Policy statement, simple expressions may be combined together using the boolean operators **and** (**&&**) and **or** (||). These operators are preserved in the translation from PICSRules to C. This is demonstrated in the complete example of a profile translation at the end of this section.

It should be noted that Policy clauses may be assigned the value of **otherwise** rather than an RSACi expression. The expression **otherwise** is always true, and therefore is simply translated into a true constant (**1**) in the C code.

Step 2 – Simple Expression Conversion:

Each simple expression is translated into C according to a translation table.

Expression	Translation
rsaci.v < 0	datatable[currentlabel][0]
rsaci.v > 0	datatable[currentlabel][1]
rsaci.v = 0	datatable[currentlabel][2]
rsaci.v <= 0	datatable[currentlabel][3]
rsaci.v >= 0	datatable[currentlabel][4]
rsaci.v < 1	datatable[currentlabel][5]
...	...

Table 3 – Sample Conversion Table (Profiles)

As described in Section 3.4, there are 100 possible simple expressions in the RSACi rating service. Each expression is assigned a single entry in this conversion table. This table has the same ordering of simple expressions as does Table 1 in Section 3.4. It is the shared organization of these tables that allows the preprocessed profiles to properly make use of the preprocessed labels.

The conversion table is constructed by simple computations on the PICSRules expressions. Every expression appears in the form *category relop constant*. Every translation appears in the form **datatable[currentlabel][***value***].** The *value* is calculated from the three portions of the expression. Each *value* is the sum of three integers, determined by Tables 4-6 below.

Category	Value
rsaci.v	0
rsaci.s	25
rsaci.n	50
rsaci.l	75

Table 4 – Category Values

Relop	Value
<	0
>	1
=	2
<=	3
>=	4

Table 5 – Relational Operator Values

Constant	Value
0	0
1	5
2	10
3	15
4	20

Table 6 – Constant Values

Using Tables 4-6, it is simple to construct the entire conversion table for the RSACi rating service. For reference, it is included in Appendix B.

As an example, the expression **(rsaci.l <= 2)** is converted by adding the values for **rsaci.l** (75), **<=** (3), and **2** (10) to produce 88. This can be verified by the table in Appendix B.

Step 3 – Statement Construction:

In this step the expression translations from the second step are inserted into the C code framework generated in the first step.

Step 4 – Repeat for each Policy:

Steps 1 through 3 repeat until each Policy clause has been translated. Note that the first translated Policy clause will not begin with **else**. All subsequent Policy clauses will begin with **else**. This allows the profile, when evaluated, to skip over Policy clauses if a preceding clause has already triggered.

Step 5 – Default Behavior Supplied:

The last line of each translated profile is always the same.

else result[currentlabel][usernumber] = 1;

This line is always included to represent the default behavior of the profile. As stated in the PICSRules specification [PICS97], if a label does not trigger any of the existing Policy clauses, it must be accepted by the profile. The line included above is a C translation of this behavior.

Example of Profile Preprocessing

Original Profile:

```
1  (PicsRule-1.1
2    (
3    Name (Rulename "user1.rlz"
4      Description "user1.rlz created by RulesGenerator")
5    Source (SourceURL "http://title.w3.org/"
6      CreationTool "RulesGenerator/0.40"
7      Author "dshapiro@w3.org"
8      LastModified "1997-12-04T12:13-0500")
9    Serviceinfo (Name "http://www.rsac.org/"
10      Shortname "rsaci"
11      BureauURL "http://www.rsac.org/ratingsv01.html"
12      BureauUnavailable "PASS")
13    Policy (RejectUnless "(rsaci.s = 0)")
14    Policy (RejectIf "((rsaci.n > 1) or (rsaci.l > 3))")
15    Policy (AcceptIf "(rsaci.v < 2)")
16    )
17 )
```

Generated C code:

```
1  /*user1.rlz*/
2  if (!(datatable[currentlabel][27]))
3  result[currentlabel][user1] = 0;
4  else if ((datatable[currentlabel][56]) ||
(datatable[currentlabel][91]))
5  result[currentlabel][user1] = 0;
6  else if (datatable[currentlabel][10])
7  result[currentlabel][user1] = 1;
8  else result[currentlabel][user1] = 1;
```

Line numbers have been included in both examples for legibility. They are not actually part of either the profile text or the C source code.

The C code reproduced here is an example of a single profile evaluator within a C source file. The actual profile executable is made up of several different pieces. First, there are some procedures for reading the label files, generating output for the users, and controlling the profile evaluators. They reside in their own source code files. The rest of the executable consists of all of the profile evaluators. Each page of 200 profiles is compiled to an object file which is linked into the resulting executable.

The code to load the labels is run only once for each processing session. In order to keep the program's memory overhead to a minimum, labels are read into the program in pages, similar to the way that profiles were read into the preprocessor in pages. A page of 200 labels is loaded by the entry code of the executable, and all profiles evaluate against those labels before the next set of labels is loaded.

When a page of labels is loaded into the processor, the two-dimensional array **datatable** stores the preprocessed values of the label The first dimension of **datatable** corresponds to the label number. The second dimension corresponds to a particular value within the label data. The **datatable** array is loaded in a predetermined order, according to the numbers assigned to each possible RSACi simple expression, as found in the conversion table. This is the same conversion table that was used during the conversion of the profile from PICSRules to C. The exact way in which the conversion table works is covered in more detail in the previous section.

The actual label data is loaded into the **datatable** array simply by reading the preprocessed label files in order. The label preprocessor uses the same conversion table as the profiles, and turns each label into a string of ones and zeros (as shown in Section 3.4).

In the profile given as an example above, you can see that **(rsaci.s = 0)** has been converted to array entry 27, **(rsaci.n > 1)** has been converted to 56, **(rsaci.l > 3)** has been converted to entry 91, and **(rsaci.v < 2)** has been converted to array entry 10, all according to the conversion mechanism detailed in step 2 earlier in this section.

Each page of evaluators contains a loop which causes every profile in the page to evaluate against every currently loaded label. The integer **currentlabel**, the other **datatable** array index, is used to keep track of the current label being evaluated. A single label is run against all of the profile evaluators within that page before the next label begins.

The Policy clauses within a PICSRules profile summarize the information that controls which labels will be accepted by the filter and which will be blocked. In this example the profile has three Policy clauses, lines 13-15. When evaluating a PICSRule each Policy clause is examined in the order in which it appears. Whenever a Policy stipulates that a label is to be blocked or passed, that action happens immediately and evaluation of that profile is over (for the current label). If the predicate of a clause is not met, that clause has no effect on the outcome and the next Policy clause in sequence is evaluated. Should a label pass through all of the Policy clauses without explicitly causing a block or accept action then the label is accepted by default.

Line 13 indicates that the filter should block any page unless it has a sex rating of zero. This translates into lines 2-3 of the C code. The **result** array stores a zero indicating a block only if the **if** statement on line 2 holds. The **result** array is a two dimensional array, much like the **datatable** array. The first dimension of the **result** array is the number of the current label, the second dimension is the number of the current profile. With this arrangement, it is easy to read out the results from the array once computation is completed. Since the **datatable** array includes the values of evaluating all possible RSACi simple expressions for this label (determined at label preprocessing time), when the C code is run it only has to look inside an array (**datatable**) to determine the values of evaluating simple expressions for a particular label, it does not have to compute them. The proper values were loaded into the array from the preprocessed labels by the entry code of the current executable.

Line 14 of this example causes the filter to reject any label that has a nudity level greater than one or a language level greater than three. The C code on lines 4-5 is the translation of this clause.

Line 15 permits the filter to accept pages that have a violence rating of less than two. Lines 6-7 in the C code correspond to this. Note that in this case, on line 7, a one is stored in the **result** array, indicating that a label has passed.

Line 8 of the C code makes explicit the notion that a label that is not handled by any existing Policy clause is to be accepted, and thus puts a one into the **result** array.

3.5.3 Step 3 – Compilation

Once the Java-based preprocessor is finished, the Makefile that is produced during profile conversion is run. This invokes the C compiler and linker on all of the C source code files that were generated in the second step. Each file produces its own object file, to be linked together. The resulting executable is run when actual processing occurs.

3.6 Profile Processing

Activating the profile store once the preprocessing stage is over is quite simple. Since a single executable is created during profile preprocessing, that program simply needs to be run in order to begin the actual processing within the profile store. The executable will search a predetermined directory for a specific number of preprocessed label files. All variables such as the number of labels, the names of the label files and the directory to search were set during preprocessing. The preprocessor that generated the labels and profiles allows the system administrator to change these settings if necessary.

Once invoked, the executable will read the first page of label files. The contents of each file are loaded into internal arrays. Processing begins after the arrays are completely loaded into the evaluators. Each evaluator determines which labels match the profile that it represents. When all of the evaluators in the current executable have finished calculating against a set of labels, the output generator, if present, creates (or appends to) an HTML file for each user containing a list of URLs as a list of links. Each URL on a particular list corresponds to a particular user's profile. This process of "Read a page of labels, Evaluate, and Generate output" is repeated until all of the labels have been processed. A sample main procedure from a profile store executable is included in Appendix C. The code in Appendix C shows how the primary loop within the executable is organized.

To run additional labels against an already preprocessed set of profiles only the labels need to be preprocessed. Profile sets can remain untouched unless the system administrator wants to make changes (to support more users, for example). Changing the number of labels that the system evaluates at a time is simple: it requires changing one constant in the C source of the label loader, recompiling only the entry code, and relinking the executable.

4 Experiments

This chapter contains all of the data gathered from running tests on the working prototype of the system. These results will show that the profile store design far surpasses the original performance goals.

4.1 Experiment Goals

The original processing goal of this project was to create a system that was able to process 10,000 labels against 100,000 profiles in an eight hour period. As stated in Chapter 3, this goal was taken directly from the results of similar experiments done in [BM94]. This Chapter demonstrates how the system surpassed that goal. Section 4.2 contains performance data gathered in my experiments. For reference, Section 4.3 contains data other than the actual processing time, such as the preprocessing times for profiles and labels.

All PICS labels that were used during these experiments contained only RSACi ratings. All PICSRules profiles contained only Policy statements that relied on RSACi ratings. None of the URL matching abilities of PICSRules were used during these tests. All profiles and labels were randomly generated. I constructed random generators that could produce any number of valid profile or label files for any given PICS rating service.

These processing times do not include the time for output generation. Since creating a highly optimized output generator was not the focus of this thesis, I disabled the simple output generator that I had built while performing these tests.

All testing was done using Sun's JDK version 1.1.5, running under WinNT 4.0. The C compiler used was Microsoft's 32-bit C/C++ compiler, version 11.00.7022. For hardware, I used a dual Pentium Pro 200 workstation with 128MB of RAM. While processing tests were running, there were no other CPU intensive processes running in the foreground on the same CPU.

All times given in this Chapter are recorded as seconds.

4.2 Processing Time

Table 7 contains time values that were recorded directly from the system. No calculation or extrapolation was used in order to obtain these numbers.

	5,000 Labels	10,000 Labels	15,000 Labels	20,000 Labels
10,000 Profiles	32	69	102	147
20,000 Profiles	41	87	130	180
30,000 Profiles	49	112	165	229
40,000 Profiles	61	127	193	267
50,000 Profiles	71	157	224	297

Table 7 – Processing Times

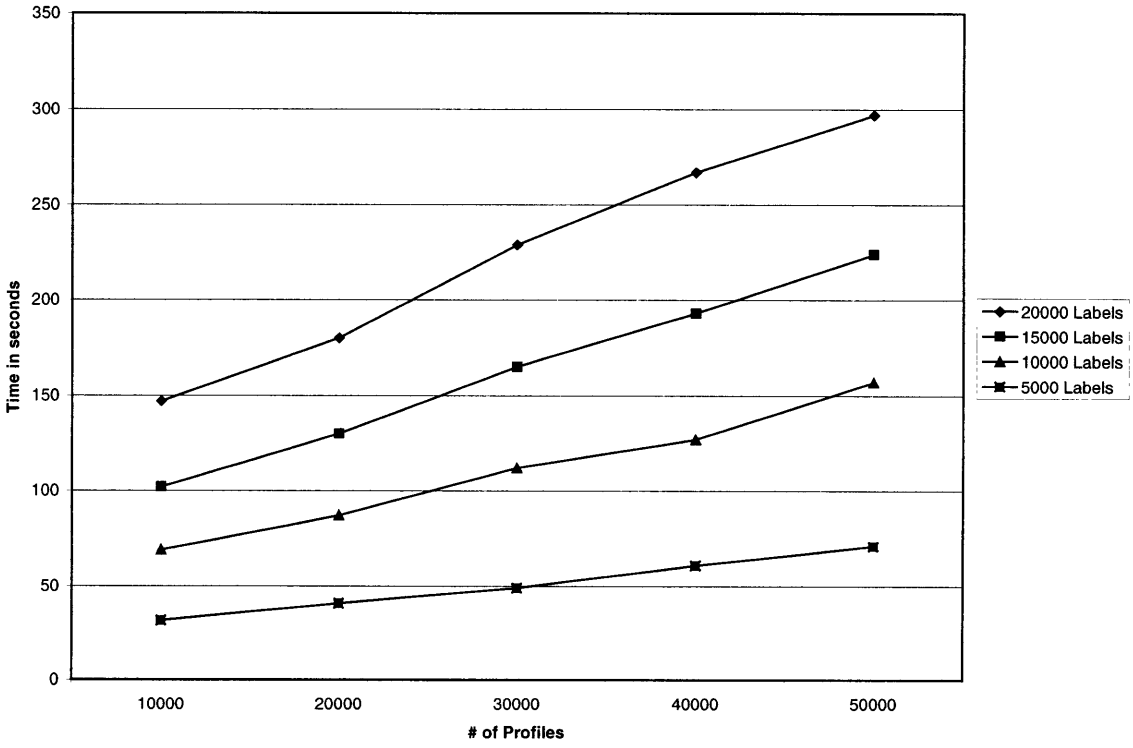

Figure 5 – Processing Time (1)

From these values, results for large-scale datasets were computed. For details on how these computations were made, see Section 4.4 "Calculations."

The numbers from Table 7, as plotted in Figures 5 and 6, show that the overall performance of the system was linear in both parameters. Regardless of whether the changes were to the number of labels or to the number of profiles, the overall shape of the data was linear.

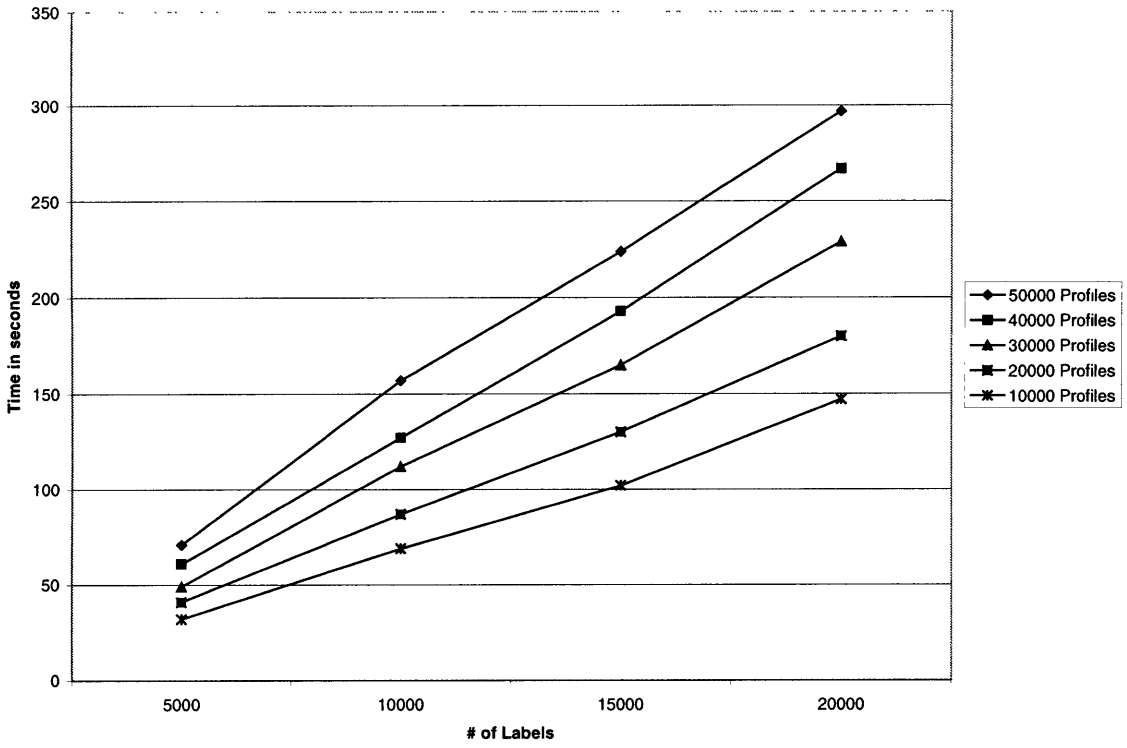

Figure 6 – Processing Time (2)

The end result of computing the system's performance on large data sets was that in an eight hour period, approximately 1.1 million labels can be processed against 100,000 profiles. This is over 100 times as many labels as the original design goal.

4.3 Other Collected Data

For reference, Tables 8 and 9 include some measurements taken on other components of the system. As you can see, the time required for label preprocessing is not significant. At the rates shown in Table 8, 10,000 labels could easily be preprocessed in under twenty minutes.

Profile preprocessing is much more complicated, and hence takes longer. Both the time it takes to convert the profiles to C code (preprocessing) and the time it takes to compile the C code are shown. Although profile preprocessing is a fairly expensive operation, it should be noted, as it was in Chapter 3, that it does not need to be done every time labels are to be evaluated. Compiled profiles can be given new labels endlessly. More profile preprocessing only has to be done when more, new profiles are added to the overall data set. Furthermore, adding new profiles to the profile set does not require repreprocessing existing profiles. Only the new profiles need to be converted and compiled.

1,000 Labels	2,000 Labels	3,000 Labels
98	177	277

Table 8 – Label Preprocessing

	10,000 Profiles	20,000 Profiles	30,000 Profiles
Preprocessing	780	1560	2400
Compilation	35	66	99
Total	815	1626	2499

Table 9 – Profile Preprocessing

4.4 Calculations

Included below are two different sets of calculations that I performed. Both sets of computations are based on the data collected in Table 7. The first set is a test to see the behavior of profiles while holding the label count constant. The second set of data was to determine how changing the label count affected the rate at which profiles could evaluate labels.

4.4.1 Profile Measurements

Table 10 contains calculated results for the four different label sets.

	5,000 Labels	10,000 Labels	15,000 Labels	20,000 Labels
Average (measured) time for 10,000 more Profiles	10	22	31	38
Linear fit to time for 10,000 more Profiles	10	20	30	40
# of Profiles possible in 8 hours	28,800,000	14,400,000	9,600,000	7,200,000

Table 10 – Profile Linearity

The results from Table 10 indicate that when the label size was held constant, the profile set exhibited linearity. That is to say, the amount of time that it took to evaluate an arbitrary number of profiles against a fixed set of labels was linearly dependent on the number of profiles.

These calculations also presented the first evidence that the labels also scaled linearly, given that the amount of time that it took to evaluate an additional 10,000 profiles scaled linearly for each additional 5,000 labels.

4.4.2 Label Measurements

The observed behavior of changes to the label set size was linear. For each step of 5,000 more labels, the evaluation rates remained linear, as shown in Figures 5 and 6 in Section 4.2. Table 11 shows the average time that it took to add 5,000 more labels to the different profiles sets.

	Average time for 5,000 more labels
10,000 Profiles	38
20,000 Profiles	46
30,000 Profiles	60
40,000 Profiles	69
50,000 Profiles	75

Table 11 – Label Linearity

As you can see in Table 11, adding 5,000 labels to the processor increased the total processing time in a manner dependent on the number of profiles. Since the behavior of the system was linear with respect to profile count, and the system was linear with respect to label count for any number of profiles, the total system scaled well in both directions.

4.5 Experiment Summary

Many tests were run on the system using datasets consisting of up to 50,000 profiles and 20,000 labels. From these experiments, the performance results of much larger data sets could be computed, since the measured experiments proved that this system behaved in a linear manner with respect to both the number of profiles and the number of labels. From linear fits to the data it was deduced that this system is over 100 times faster than the original design goal of 10,000 labels vs. 100,000 profiles in 8 hours. Extrapolation of the existing performance data resulted in the system being able to process 1.1 million labels against 100,000 profiles in 8 hours.

5 Future Research

There are several areas in which the system I designed could be further expanded and improved. These fall into three general categories: filtering options, output quality, and performance.

5.1 Filtering

There are many different options that could be added to the filtering mechanism as future projects. Listed in this section are just a few of the more interesting possibilities.

1. Label Bureau Access – Instead of providing labels directly to the profile store, the system could be modified so that it contacts label bureaus directly to get PICS labels.
2. DSig Authorization – Labels possessing digital signatures could be used to verify the authenticity of the label's information.
3. Multiple PICS Rating Services – Currently, the system only supports the RSACi rating service. A useful addition would be to provide support for other popular PICS services such as SafeSurf. Eventually support for arbitrary PICS rating systems would be ideal. With support for different rating services, users can employ profiles that check labels of multiple rating services at once. Users should not be forced to choose a single rating service.
4. Non-label-only PICS Rating Services – The preprocessor of the system was only designed with label-only rating services in mind. The idea of precomputing all possible simple expressions against a label is not possible if the expressions can range over all possible integers or real numbers. A totally new preprocessing mechanism will need to be designed in order to provide support for non-label based PICS rating services.
5. URL Matching Filters – The system currently provides no support for the AcceptByURL and RejectByURL types of Policy clauses of a PICSRule. Adding this functionality would make the profile store fully compliant with the PICSRules specification.
6. Direct Comparison with Other Techniques – The project did not consist of any head-to-head comparisons of filtering techniques on the same set of data. An entirely different thesis-quality project could be to compare the various filtering techniques enumerated in Chapter 1 and see how they compare against each other directly in terms of speed, usability, and usefulness of results

5.2 Output

The output generated by this system could be improved. The current output generator is a severe performance bottleneck. In order to achieve the desired processing

speed, the output generation had to be disabled entirely. A fast way of generating output for user consumption is a worthy project in and of itself.

The simple output generator that I constructed created an individualized HTML file for each user profile. Many other options for user output could be created. An email based system would prevent the need for users to look at a particular Web page just to get their profile results.

Perhaps the most exciting possibility is a system in which users can specify the type of output that they want, as well as the frequency of updates. People may want information at a variety of different rates: daily, weekly, etc... All users should not have to be forced into the same data rate.

5.3 Performance

The final performance of this system, while exceeding the initial design goals, could still be improved further. More specifically, reducing the disk space and memory requirements would make this a better system. All tests were run on a rather large dual-CPU workstation. Even granted that hardware is increasing in power rapidly, the hardware demands of this system were still quite large.

Reducing the amount of time required for profile preprocessing would also be a great improvement. Although profile preprocessing does not have to be repeated frequently, the time it takes on large numbers of profile is somewhat of a hindrance. The initial time cost of setting up a system such as this for millions of users would be rather large.

6 Conclusion

This thesis has shown how to efficiently perform a very large number of user-specified queries to filter information from the Web using metadata rather than actual content. The key idea was the transformation of user profiles from their original PICSRules format into a completely machine-executable format, by way of C source code. This transformation accounts for the system's high performance, as demonstrated by earlier versions based on different designs that proved to be orders of magnitude slower.

One goal of this project was to change the way information is presented to users from the traditional Web pull-based browsing model to a push-based channel paradigm. The technique described here shows that the push model can scale to millions of channels, each one representing a user receiving personal selections from hundreds of thousands of new data items daily. This transforms the Web from an endless morass of content through which users must search into a personalized content channel that contains only information that matches the user's preferences. In the future, most users will probably expect their information to be delivered by a content channel and will only resort to today's "browsing" model for unanticipated information needs.

The performance analysis in Chapter 4 allows us to extrapolate processing times for handling all users of the Internet and all data on the Web. While estimates for both the numbers of Internet users and servers vary widely, I have chosen the following:

- Nua Internet Surveys [NUA98] estimates 112.75 million users in Feb. 1998.
- Network Wizards [NW98] estimates 29.67 million hosts in Jan. 1998.

Assuming that each site is labeled with a single generic PICS label [PICS96b] we can make the following performance predictions:

1. 100,000 user profiles can be computed against the entire Web by 9 machines running this system for a single 24 hour period.

2. 10,000 labels can be computed against all Internet users by 3 machines running this system for a single 24 hour period.

3. The entire Web can be computed against every Internet user by 1,110 machines running this system for one week.

These computations show that this technology supports channels of truly staggering proportions. The hardware on which I ran this system was only a desktop PC. Given the existence of much more powerful server machines, the rapid advances in hardware design, and further refinements to this system, the ability to filter the entire Web and continuously deliver to each user custom filtering results is a realistic feat for the near future.

Appendices

Appendix A RSACi Rating Service

```
((PICS-version 1.1)
 (rating-system "http://www.rsac.org/ratingsv01.html")
 (rating-service "http://www.rsac.org/")
 (name "The RSAC Ratings Service")
 (description "The Recreational Software Advisory Council rating service. Based on the work of Dr.
 Donald F. Roberts of Stanford University, who has studied the effects of media on children for nearly 20
 years.")
 (default (label-only true))

 (category
  (transmit-as "v")
  (name "Violence")
  (label
   (name "Conflict")
   (description "Harmless conflict; some damage to objects")
   (value 0))
  (label
   (name "Fighting")
   (description "Creatures injured or killed; damage to objects; fighting")
   (value 1))
  (label
   (name "Killing")
   (description "Humans injured or killed with small amount of blood")
   (value 2))
  (label
   (name "Blood and Gore")
   (description "Humans injured or killed; blood and gore")
   (value 3))
  (label
   (name "Wanton Violence")
   (description "Wanton and gratuitous violence; torture; rape")
   (value 4)))

 (category
  (transmit-as "s")
  (name "Sex")
  (label
   (name "None")
   (description "Romance; no sex")
   (value 0))
  (label
   (name "Passionate kissing")
   (description "Passionate kissing")
   (value 1))
  (label
   (name "Clothed sexual touching")
   (description "Clothed sexual touching")
   (value 2))
  (label
```

```
 (name "Non-explicit sexual activity")
 (description "Non-explicit sexual activity")
 (value 3))
(label
 (name "Explicit sexual activity; sex crimes")
 (description "Explicit sexual activity; sex crimes")
 (value 4)))

(category
 (transmit-as "n")
 (name "Nudity")
 (label
 (name "None")
 (description "No nudity or revealing attire")
 (value 0))
 (label
 (name "Revealing Attire")
 (description "Revealing attire")
 (value 1))
 (label
 (name "Partial Nudity")
 (description "Partial nudity")
 (value 2))
 (label
 (name "Frontal Nudity")
 (description "Non-sexual frontal nudity")
 (value 3))
 (label
 (name "Explicit")
 (description "Provocative frontal nudity")
 (value 4)))

(category
 (transmit-as "l")
 (description "Language")
 (label
 (name "Slang")
 (description "Inoffensive slang; no profanity")
 (value 0))
 (label
 (name "Mild Expletives")
 (description "Mild expletives")
 (value 1))
 (label
 (name "Expletives")
 (description "Expletives; non-sexual anatomical references")
 (value 2))
 (label
 (name "Obscene Gestures")
 (description "Strong, vulgar language; obscene gestures")
 (value 3))
 (label
 (name "Explicit")
 (description "Crude or explicit sexual references")
 (value 4))))
```

Appendix B Conversion Table (Profiles)

Expression	Translation	Expression	Translation
rsaci.v < 0	datatable[currentlabel][0]	rsaci.n < 0	datatable[currentlabel][50]
rsaci.v > 0	datatable[currentlabel][1]	rsaci.n > 0	datatable[currentlabel][51]
rsaci.v = 0	datatable[currentlabel][2]	rsaci.n = 0	datatable[currentlabel][52]
rsaci.v <= 0	datatable[currentlabel][3]	rsaci.n <= 0	datatable[currentlabel][53]
rsaci.v >= 0	datatable[currentlabel][4]	rsaci.n >= 0	datatable[currentlabel][54]
rsaci.v < 1	datatable[currentlabel][5]	rsaci.n < 1	datatable[currentlabel][55]
rsaci.v > 1	datatable[currentlabel][6]	rsaci.n > 1	datatable[currentlabel][56]
rsaci.v = 1	datatable[currentlabel][7]	rsaci.n = 1	datatable[currentlabel][57]
rsaci.v <= 1	datatable[currentlabel][8]	rsaci.n <= 1	datatable[currentlabel][58]
rsaci.v >= 1	datatable[currentlabel][9]	rsaci.n >= 1	datatable[currentlabel][59]
rsaci.v < 2	datatable[currentlabel][10]	rsaci.n < 2	datatable[currentlabel][60]
rsaci.v > 2	datatable[currentlabel][11]	rsaci.n > 2	datatable[currentlabel][61]
rsaci.v = 2	datatable[currentlabel][12]	rsaci.n = 2	datatable[currentlabel][62]
rsaci.v <= 2	datatable[currentlabel][13]	rsaci.n <= 2	datatable[currentlabel][63]
rsaci.v >= 2	datatable[currentlabel][14]	rsaci.n >= 2	datatable[currentlabel][64]
rsaci.v < 3	datatable[currentlabel][15]	rsaci.n < 3	datatable[currentlabel][65]
rsaci.v > 3	datatable[currentlabel][16]	rsaci.n > 3	datatable[currentlabel][66]
rsaci.v = 3	datatable[currentlabel][17]	rsaci.n = 3	datatable[currentlabel][67]
rsaci.v <= 3	datatable[currentlabel][18]	rsaci.n <= 3	datatable[currentlabel][68]
rsaci.v >= 3	datatable[currentlabel][19]	rsaci.n >=3	datatable[currentlabel][69]
rsaci.v < 4	datatable[currentlabel][20]	rsaci.n < 4	datatable[currentlabel][70]
rasci.v > 4	datatable[currentlabel][21]	rsaci.n > 4	datatable[currentlabel][71]
rsaci.v = 4	datatable[currentlabel][22]	rsaci.n = 4	datatable[currentlabel][72]
rsaci.v <= 4	datatable[currentlabel][23]	rsaci.n <= 4	datatable[currentlabel][73]
rsaci.v >= 4	datatable[currentlabel][24]	rsaci.n >= 4	datatable[currentlabel][74]
rsaci.s < 0	datatable[currentlabel][25]	rsaci.l < 0	datatable[currentlabel][75]
rsaci.s > 0	datatable[currentlabel][26]	rsaci.l > 0	datatable[currentlabel][76]
rsaci.s = 0	datatable[currentlabel][27]	rsaci.l = 0	datatable[currentlabel][77]
rsaci.s <= 0	datatable[currentlabel][28]	rsaci.l <= 0	datatable[currentlabel][78]
rsaci.s >= 0	datatable[currentlabel][29]	rsaci.l >= 0	datatable[currentlabel][79]
rsaci.s < 1	datatable[currentlabel][30]	rsaci.l < 1	datatable[currentlabel][80]
rsaci.s > 1	datatable[currentlabel][31]	rsaci.l > 1	datatable[currentlabel][81]
rsaci.s = 1	datatable[currentlabel][32]	rsaci.l = 1	datatable[currentlabel][82]
rsaci.s <= 1	datatable[currentlabel][33]	rsaci.l <= 1	datatable[currentlabel][83]
rsaci.s >= 1	datatable[currentlabel][34]	rsaci.l >= 1	datatable[currentlabel][84]
rsaci.s < 2	datatable[currentlabel][35]	rsaci.l < 2	datatable[currentlabel][85]
rsaci.s > 2	datatable[currentlabel][36]	rsaci.l > 2	datatable[currentlabel][86]
rsaci.s = 2	datatable[currentlabel][37]	rsaci.l = 2	datatable[currentlabel][87]
rsaci.s <= 2	datatable[currentlabel][38]	rsaci.l <= 2	datatable[currentlabel][88]
rsaci.s >= 2	datatable[currentlabel][39]	rsaci.l >= 2	datatable[currentlabel][89]

rsaci.s < 3	datatable[currentlabel][40]	rsaci.l < 3	datatable[currentlabel][90]
rsaci.s > 3	datatable[currentlabel][41]	rsaci.l > 3	datatable[currentlabel][91]
rsaci.s = 3	datatable[currentlabel][42]	rsaci.l = 3	datatable[currentlabel][92]
rsaci.s <= 3	datatable[currentlabel][43]	rsaci.l <= 3	datatable[currentlabel][93]
rsaci.s >= 3	datatable[currentlabel][44]	rsaci.l >= 3	datatable[currentlabel][94]
rsaci.s < 4	datatable[currentlabel][45]	rsaci.l < 4	datatable[currentlabel][95]
rsaci.s > 4	datatable[currentlabel][46]	rsaci.l > 4	datatable[currentlabel][96]
rsaci.s = 4	datatable[currentlabel][47]	rsaci.l = 4	datatable[currentlabel][97]
rsaci.s <= 4	datatable[currentlabel][48]	rsaci.l <= 4	datatable[currentlabel][98]
rsaci.s >= 4	datatable[currentlabel][49]	rsaci.l >= 4	datatable[currentlabel][99]

Appendix C Sample Profile Evaluator Main Procedure

```
int se_table[200][100];
char labelurl[200][400];
int lcount = 10000;
int vmsize = 200;

main(int argc, char *argv[])
{
int b = 0;
int d = 0;
for (b=0; b<lcount/vmsize; b++) {
loadtable(labelurl, se_table, d, vmsize);
procgroup_1(labelurl, se_table, vmsize);
procgroup_2(labelurl, se_table, vmsize);
```

... additional calls to procgroup procedures appear here

```
d = d + vmsize;
printf("Finished %d labels\n", d);
}
}
```

References

[BM94] Beebee, P., Miller, J. (1994), "Query Processing for Information Distribution," March 16. 1994.

[DRP97] "The HTTP Distribution and Replication Protocol," *W3C Submission from Marimba, Netscape, Sun Microsystems, Novell, and @Home*, August 25, 1997 available from *http://www.w3.org/TR/NOTE-drp-19970825.html.*

[KSK97] Kamba, T., Sakagami, H., Koseki, Y. (1997), "Automatic Personalization on Push News Service," *W3C Workshop on Push Technology*, September 8, 1997.

[NUA98] "Nua Internet Surveys," available from: *http://www.nua.ie/surveys/how_many_online/index.html.*

[NW98] "Network Wizards," available from: *http://www.nw.com/zone/WWW/report.html.*

[PICS96a] "Rating Services and Rating Systems and Their Machine-Readable Descriptions Version 1.1," *W3C Recommendation,* October 31, 1996, available from *http://www.w3.org/TR/REC-PICS-services.*

[PICS96b] "Label Distribution Label Syntax and Communication Protocols Version 1.1," *W3C Recommendation,* October 31, 1996, available from *http://www.w3.org/TR/REC-PICS-labels.*

[PICS97] "PICSRules 1.1," *W3C Recommendation,* December 29, 1997, available from *http://www.w3.org/TR/REC-PICSRules.*

[PICSLG] "PICS LabelGrabber," *W3C Reference Code*, January 30, 1998, available from *http://www.w3.org/PICS/refcode/LabelGrabber/index.htm.*

[PICSPC] "ProfileCreator," *W3C Reference Code*, August 1 1997, available from *http://www.w3.org/PICS/Profiler/.*

[PICSSL] "PICS Standard Library," *W3C Reference Code,* August 22 1997, available from *http://www.w3.org/PICS/refcode/Parser/.*