Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

2003

Aspect-Oriented Programming (AOP): Dynamic Weaving for C++

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Yan CHEN

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Jean-Marc Menaud (Ecole des Mines de Nantes)

Abstract

Due to the fast development of Internet services and the huge amount of network traffic, it is becoming an essential issue to reduce its user-perceived latency. Although Web performance is improved by caching, cache can provide more benefit than nowadays: prefetching reduces user access time, at the expense of increasing traffic. Performance measurement of prefetching techniques are made primarily in terms of hit ratio and bandwidth usage. A significant performance factor for a prefetching algorithm in its ability to reduce latency is deciding which objects to prefetch in advance. Nevertheless, to achieve the best performance, the prefetching policy must match user access patterns and Web server characteristics. This implies that new prefetching policies must be loaded dynamically as needs change.

Recent Web caches are large C++ programs, and thus adding a single prefetching policy to an existing Web cache is a daunting task. Providing multiple policies is even more complex. The essential problem is that prefetching concerns crosscut the cache structure. Aspect-Oriented Programming (AOP) is a natural technique to address this issue. Actually, μDyner, a Dynamic Weaver on native code address to provide dynamic weaving of aspects targeted toward C applications. It provides lower overhead for aspect invocation than other dynamic approaches, thus meeting the performance needs of Web caches [22]. But, the unique main language supported by μDyner is the C language.

The main goal of this dissertation is to realize a study on a dynamic weaving mechanism for the C++ language and upgrade μDyner to support this new feature.

Keyword: Aspect-Oriented Programming, AOP, dynamic weaving, C++, prefetching, Web caches

Acknowledgements

Firstly, I would like to express my sincere appreciation to Professor Jean-Marc Menaud for his guidance and support.

Pure-hearted appreciation goes to Marc Ségura-Devillechaise who helped me a lot during this year.

Many thanks go to my parents for their substantial and mental support. ”I learned a lot!” what I can say to them as a conclusion for this happy and hard year. I want to share my every memorable moment with them.

Also, a big thank goes to Jacques Noyé and all the professors of EMOOSE.

Last, thanks to one of my best old friends - Xiaoyu CHEN for his help and encourage in the hardest period. Thanks to all EMOOSErs who make the life more wonderful. Thanks to my new friends and every one who cares for me.

Yan CHEN

Nantes, France
August 2003

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web traffic now accounts for most of the Internet traffic. Caching is one way to decrease the amount of traffic: caches move documents close clients that need them. Thus, caches reduce the number of server requests. Unfortunately, recent results suggest that the maximum achievable cache hit rate is close to 40% to 50% [15]. Regardless of the caching scheme in use, one out of two documents cannot be found in the cache.

One way to increase the caching hit ratio is to *anticipate* future documents requests and *preload* or *prefetch* these documents in the cache. Prefetching transfers Web contents in the cache before they have been requested in the hope that they will be accessed into near future. An appropriate prefetching scheme can thus potentially decrease the latency experienced by users [18]. In order to balance the reduction of the latency experienced by users against the additional traffic on the network, prefetching strategies need to be tailored to user access pattern and to Web server characteristics. To support the use of such policies, a Web cache must be able to load or unload the new policies on demand.

Many Web caches, such as Squid [24], are implemented using static module architecture, fixed at compile time. The common approach to extend such architecture relays on loading a new module implementing a new functionality. Unfortunately, prefetching policies crosscut the cache structure [22]: module loading is this in adequate. Aspect-Oriented Programming (AOP) appears as the most appropriate solution.

There is already one AOP system: μDyner targeting the adaptation needs of Web caches. μDyner is an Aspect-Oriented Programming system for writing and dynamically deploying aspects in running C programs without program interruption. But future versions of the most commonly used Web caches - Squid will be written in C++.

Despite the fact that C++ is a superset of C, C and C++ are very different languages. The differences range from the source level up to the binary level. The goal of this dissertation is to explore, if a tool similar to μDyner for C can be devised for C++.

In other words, the aim of this dissertation is twofold. Its first purpose is to determine if the weaving techniques used in μDyner are appropriate for C++. Its second purpose is to discuss an aspect language for C++.

The rest of this dissertation is structured as follows:

- **Chapter Two** explains the motivation of this dissertation. It points out the crosscutting nature of prefetching policies. Considering different algorithms of prefetching policies, this chapter analyzes how those algorithms crosscut the typical web cache - Squid.
- **Chapter Three** describes the different aspects systems available. It concludes that no aspect system provides dynamic weaving for C++.
- **Chapter Four** illustrates the solution we propose. It studies how binary rewriting techniques can be applied to C++ to build an aspect system.
- **Chapter Five** describes our implementation.
- **Chapter Six** summarizes and concludes before describing some future work.

Chapter 2

Motivation

This dissertation is motivated by the need to integrate prefetching policies on the fly in a Web cache. This chapter firstly details what is a Web cache and presents the structure of the most commonly used Web cache: Squid. Then this chapter continues with a description of prefetching before showing why prefetching is a crosscutting concerns of Web cache.

2.1 Web Caches

Internet is becoming de facto standard to disseminate scientific, commercial and personal information. The Web consists of individual pages linked to or from other pages through Hyper Text Markup Language (HTML) constructs. Although there are different protocols, HTML over HTTP amounts to 80% [8] of the Web traffic. The Web offer us a quick access to tremendous variety of information from anyplace in the world. We have all experienced that there is a delay between moment when we request a Web object and the moment when the browser starts to load the object on server. Such too high latency discourages us using the Web. In fact, reducing this latency is an essential problem both for the content provider and the Internet Service Provider (ISP) to attract and keep customers. Web caches are a low cost software solution. That can reduce the network traffic and the server workload.

Basically, a cache is a kind of software that stores commonly accessed data elements on disks. A Web cache acts as an intermediary between users and Web servers. Upon reception of a request, if a local copy of the requested document is available, the Web cache will return this copy to the user. Otherwise, it will relay the request to the Web server. The cache will keep a copy of the newly downloaded document and send it to the user. A number of analytical and empirical studies have already established the effectiveness of this approach [4, 5, 11, 25]. To further increase efficiency, a Web cache can share its local space with others caches. This approach is known as cooperative caching.

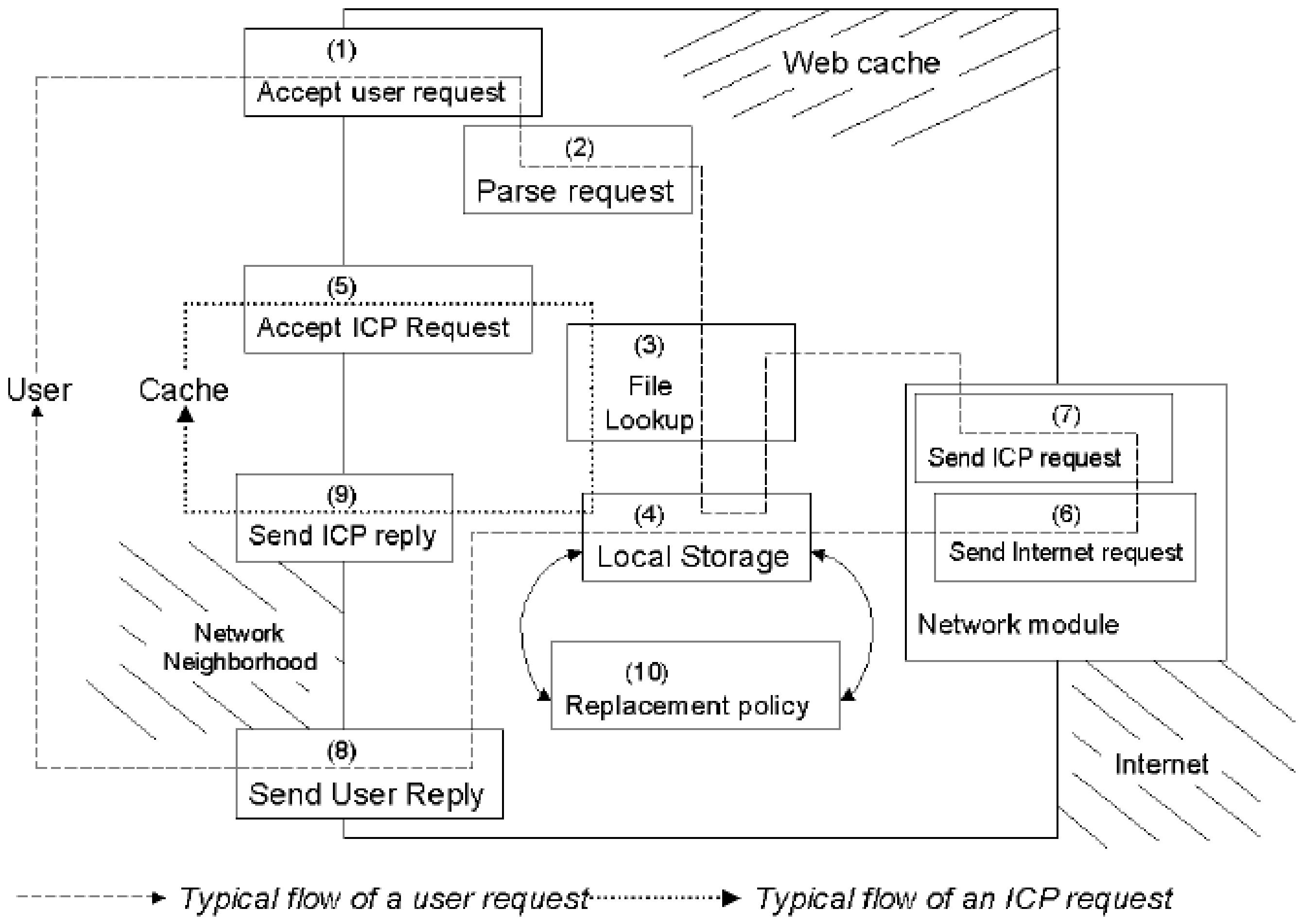

Figure 2.1: Typical structure of a Web cache

2.1.1 Typical structure of a Web cache

Typical Web caches are structured into modules, each one implementing a different concern. Figure 2.1 presents the structure of the most widely used web cache: Squid, an open-source Web cache developed by the National Laboratory for Applied Network Research and the members of the Web community [24]. The main Squid software functionalities are split in different modules. As illustrated in Figure 2.1, these modules realize three basic tasks: user request management, interaction with the neighbors and local storage management. To process a user request, the cache accepts and parses the request (1 and 2), searches the document in the local storage (3 and 4), and possibly forwards the request to the Web server (6) or to the neighbors it is cooperating with (7). If a new document is obtained, it is saved to the local storage (4), which may require the activation of the replacement policy (10) to free some space. Finally, the document is sent to the user (8). Upon reception of a cooperation request from another cache in the neighborhood, the local cache checks whether the requested document is locally available (3 and 4), and answers with either an error message or the requested document (9).

2.2 Prefetching

Only 22% of the resources are accessed more than once on the Web [18]. Hopefully, these 22% accounts for 50% of the requests. One way to further increase the cache hit ratio is to anticipate future requests and prefetch these objects into a local cache [26]. Studies of traces have shown that [14]:

- Caching at the proxy side alone could reduce latency by at best 26%
- Prefetching at the proxy side could reduce latency by at best 57%
- Prefetching and caching at the proxy side could lead to a maximal 60% reduction of the perceived latency

When making good predictions about future requests, the cache receives pages that are highly likely to be accessed soon. Thus, the performance of the cache is improved and this in turn has a direct effect in reducing the delays perceived by users [18]. In the future, it is likely that prefetching could not be ignored on the Web as microprocessor or file system designers can already not ignore it. Today, however, most Web caches do not support prefetching and HTTP provides no support for it.

2.3 Caching/Prefetching Architecture

The prefetching policies can use different kind of input such global access statistics as (1) estimation of object reference frequencies and (2) estimation of object lifetimes. These estimations can be well maintained by servers. Servers can collect user access statistics and publish information of objects popularity along with patterns of usage. They can also send prefetching hints to cache.

2.3.1 Dynamical integration at the binary level

Best performance will only be reached if the cache employs an appropriate prefetching policy. But a cache cannot predict by which client application it will be contacted. For example, it does not know if the HTML traffic will be mostly generated by browsers or robots like `wget`. Additionally, a cache does not know in advance with which server it will interact. Thus, the cache cannot choose at compile time the correct prefetching policy because it does not know which policy will be the most appropriate. That's why the integration has to be dynamic. Previous work [17] has shown the difficulty to integrate a new feature in Squid. For example, the replacement of Squid cooperation protocol - protocol already localized into two modules (roughly 10% of the code) - required in practice the alteration of seven modules representing roughly 40% of the code [16]. Moreover stopping the cache during the integration is not possible because the integration of a new policy must be done without losing the cache content. It must be done through code downloaded from servers and clients and executed in the cache. Therefore, Aspect-Orientation has been proposed as integration technology. The approach requires to be able to weave dynamically on the native code of a running processes.

2.4 Prefetching Policies

A prefetching policy is the combination of two algorithms: a prediction algorithm and a threshold algorithm. The prediction algorithm is responsible of producing a list of Web objects that are likely to be requested soon. Typical prediction algorithms use the user/server access history. The threshold algorithm is in charge of determining the number of documents to prefetch within this list depending on the resources available. For example, if the network link between the cache and a given server is very fast, then it makes little sense of prefetching the contents of that server: it will be better to prefetch pages from slower servers.

In this section we describe different algorithms. While looking for threshold algorithms, it appears that only the Jiang's [12] threshold algorithm has been published. The other implementations are all using a fixed hard coded threshold. Therefore this section mostly describes the principles behind different prediction algorithms.

2.4.1 Threshold Algorithm

The threshold algorithm computes the prefetch threshold for each server, which is the number of document to prefetch for each server. This threshold embodies a trade-off between bandwidth usage and latency. Zhimei Jiang [12] proposes to prefetch documents only if their access probabilities is superior to a computed threshold. They proposed the following formula as an upper bound estimation of the threshold:

$$H = 1 - \frac{(1-\rho)\dfrac{\alpha_T}{\alpha_B}}{(1-\rho)^2 b + \dfrac{\alpha_T}{\alpha_B}}$$

Where:
α_T dollars per time unit;
α_B dollars per packet;
$\rho = \lambda * x$ is the system load;
λ is the arrival rate of user request without prefetching;
x is the time unit;
b is the system file capacity

For fixed b and $\frac{\alpha_T}{\alpha_B}$, as ρ increase, the over all trend of the prefetch threshold increase as well, which means that fewer files should be prefetched. But Zhimei Jiang [12] found that if the increase of thredhold is not monotonic for small values of $\frac{\alpha_T}{\alpha_B}$, the load is low, prefetching does not save much time. As the load increase, it takes longer to transmit a file and prefetching can save more time, therefore the threshold decreases.

2.4.2 Prefetching by Partial Match

Prediction by Partial Match (PPM) tries to capture recurring access patterns [18]. The algorithm is based on calculating and comparing the probabilities of some requests occurring in the future. These probabilities are derived from the number of times each Web object has been accessed. These are some variations of this algorithm: Selecting requests from the highest possible order only; combining predictions from several orders; assigning different levels of trust to various request orders; Choosing up to k requests to prefetch, where $1 \leq k \leq k_{max}$.

Kroeger and Long [14] present an implementation of delta-compression techniques, based on Prediction by Partial Match (PPM). Through this implementation, the prediction of the future's request is decided by previous accesses k. By the trace-driven simulation, cache-hits increase averagely 25% than a non-prefetching and 4MB predictive cache could get higher hit rate than a 90MB cache with LRU. The major drawback is this algorithm's memory requirement.

Li Fan's [7] results show that prefetching combined with cache and delta-compression can reduce user-perceived latency up to 23.4%. The reduction is achieved using the Prediction by Partial Matching (PPM) algorithm, with an accuracy ranging from 40% up to 73% and generating 1% to 15% extra traffic on the modem links. A perfect predictor can increase the latency reduction to 28.5%, whereas without prefetching, large browser cache and delta-compression can only reduce latency by 14.4%. PPM can be configured to meet the desired properties.

PPM is especially appropriate for users carrying out a single type of activity and thus having very stable access patterns.

2.4.3 Prefetching by popularity

In 1998, Markatos et al. proposed a top ten approach for prefetching [26]. This idea is to keep the top ten popular documents for each web server. Later on clients or proxy servers can prefetch these documents. Their result shows that this approach can anticipate more than 40% of client requests and achieves hit ratio close to a 60% at the cost of increasing network traffic by no more than 10% in most cases.

2.4.4 Prefetching by lifetime

The lifetime of an object is the interval between two consecutive modifications of object [26]. The longer mean lifetime an object has, the less frequently it changes. The bandwidth cost to update stale objects hence decreases. If we select N objects with the longest lifetime to replicate in the local cache, we can envision the least network traffic usage. That is, this prefetching policy is to prefetch the most time-consuming N object into the cache. Of course, this prefetching policies is efficient when the Web objects having the largest lifetime as popular as well.

2.4.5 Prefetching by Good-Fetch

The goal of this algorithm proposed by Venkatatramani [26] is to balance object access frequency and object update frequency. Prefetching has the disadvantage that the prefetched objects may not end up being used or being used before it gets stale. If an object can ends up being referenced before it goes stale, it is considered as a *good fetch*. It only prefetches objects whose probability of being accessed before being modified is above a given value. As for the object i, its lifetime is l_i; its probability of being accessed is p_i; and user request arrival rate denoting how many requests arrive per second is a. Then the probability of object i to be accessed before it dies is:

$$P_{goodFetch} = 1 - (1 - p_i)^{a*l_i}$$

In this approach, $a*l_i$ is the number of requests arriving during the lifetime of object i, $(1-p_i)^{a*l_i}$ represents the probability that none of requests arriving during the lifetime of object i. Thus, $1 - (1 - p_i)^{a*l_i}$ denotes the probability of object i to be accessed before it dies. This algorithm is interesting when the Web objects being accessed are changing frequently.

2.4.6 Prefetching by APL

As above, for object i, Ying [26] assumed that the object's lifetime is l_i, p_i for the probability of being access, with user request arrival rate is a. Then, $a * p_i$ is the user request rate for object

i, and $a * p_i * l_i$ represents the number of requests for object i that arrive before it dies. Now we discuss about the object popularity - $a * p_i * l_i$. When $n > 1$, this algorithm becomes closer to *Prefetching by Popularity*, that is, higher bandwidth is used to improve the response time. If $n < 1$, this algorithm is closer to *Prefetching by Lifetime*. Thus it reduces network bandwidth usage.

The threshold-based algorithm prefetching by APL algorithm can successful balance networks resource cost and response time reduction. Using the $\frac{H^k}{B}$ ratio ($k = 5$) to measure the performance, this prefetching performance is considered good. ($\frac{H^k}{B} = \frac{(\frac{Hit_{prefetching}}{Hit_{demand}})^k}{\frac{BW_{prefetching}}{BW_{demand}}}$, where $\frac{Hit_{prefetching}}{Hit_{demand}}$ is the hit ratio improvement of prefetching over demand caching, where $\frac{BW_{prefetching}}{BW_{demand}}$ is network bandwidth increase over demand caching, where k is the power of hit ratio improvement). It is appropriate for cache experiencing high variations in the congestion of the network: the algorithm can balance the resource consumption.

2.5 Integrating Prefetching in Web caches

Many web caches, such as Squid, are implemented using a module-based architecture. A natural strategy to extend a module-based system with a new functionality is to implement this functionality as a new module [22]. Unfortunately, in the case of prefetching policy downloaded from the clients or from the servers, the problem becomes even more acute: the code implementing the policy needs to use some internals of the cache. These internals are: checking if an object is in cache; sending a request to a server and storing an object in cache [16]. There are internals because a cache does not normally offers these features separately to its clients. While these features are available in any cache, the protocol to invoke them is therefore specific to a given implementation. Web browsers cannot anticipate the caches that they will be encountered. A protocol needed to integrate prefetching in a cache requires pervasive modifications. In terms of source code, different modifications are required in many places: prefetching is a cross cutting concern. Thus, a prefetching module would have to redundantly implement in many basic cache operations.

2.5.1 Prediction by partial match

Prediction by Partial Match (PPM) depends on calculating and comparing the probabilities of some events occurring next. For the different process, they can be classified in the following categories [18]: client-base, server-based and combined client and server.

Regarding as Combined client and server PPM, which combines client and server knowledge in order to make more accurate predictions, the prefetching policy selects the documents to prefetch based on the contained information of the user, which decides that the user should initialize the information of the new document. This characteristic decides that the new document can be initialized in the Accept User request Module in the Figure 2.1. Prefetching should also be initialized when there is an incoming request for a document that is found in the cache. That is, the Send User Reply modular in Figure 2.1 interacts with roiling the modular interface.

In addition, although Themistoklis Palpanas et al. [18] use the replacement policy in Lowest Relative Value (LRV), popularly uses the LRU replacement, which depends on document's probability to choose the documents to remove from the cache. Because the basic assumption of a prefetching policy is that prefetched documents will be used shortly after the requested document, the access time associated with a prefetched document should be based on the access time of the reused document, and not on the access time of the prefetched document itself. The shorter access time is higher probability. Thus, the Local storage module in Figure 2.1 must be adapted with an aspect that identifies new prefetched documents and sets their access times accordingly. The File Lookup module in Figure 2.1 must similarly be adapted with an aspect that updates the probability of the associated prefetched documents when a requested document is reaccessed from the cache. Finally, the replacement policy itself should be adapted with an aspect that removes the associated prefetched documents when the requested document is removed from the cache.

2.5.2 Prefetching by popularity

The popularity algorithm keeps the N most popular objects in the cache and updates them immediately whenever these objects are modified. The Prefetching by Popularity policy is also based on the contained popularity information of the document. This character decides that the new document only can be initialized after the Network Module in the Figure 2.1. It will interact the Send ICP request module and Send Internet request module, at which modules the contents of the document have been sent. Prefetching should also be initialized when there is an incoming request for a document that is found in the cache. That is, the Send User Reply modular in Figure 2.1 interacts with roiling the modular interface.

Concerning the replacement policy, a LRU replacement depends on document's popularity to choose the documents to remove from the cache. As more frequently request, the access time is shorter. The same as 2.5.1, the Local storage modular in Figure 1 must be adapted with an aspect that identifies new prefetched documents and sets their popularities accordingly. The File Lookup module and the replacement policy are the same to 2.5.1.

2.5.3 Prefetching by lifetime

The Lifetime of an object is the interval between two consecutive modifications of object. With the same analysis method to the 2.5.1, the Prefetching by Lifetime policy is based on the contained Lifetime information of the document. And with this character, the new document also only can be initialised after the Network Module in the Figure 2.1. For the same reason like 2.5.1, the prefetching should only be activated by the user's request, instead of the neighbor's or prefetching's request.

When it comes to the Reply for a document that is found in the cache, the Prefetching should also be initialized. That is the same to 2.5.1, the Send User Reply modular in Figure 2.1 interacts with roiling the modular interface. With the same condition, this prefetching policy should only be applied by the user request.

Moreover, the replacement policy, a LRU replacement depends on document access time to choose the documents to remove from the cache. In this case, the Local storage modular in Figure 2.1 must be adapted with an aspect that identifies new prefetched documents and sets their access times. The File Lookup module in Figure 2.1 must similarly be adapted with an aspect that updates the lifetime. Finally, the replacement policy modular in Figure 2.1 must be adapted with an aspect that removes the associated prefetched documents when the requested document is removed from the cache, this function is the same to 2.5.1.

2.5.4 Conclusion

The crosscutting nature of prefetching policies suggests that such policies should be implemented using aspect oriented programming (AOP). Actually the wide range of prefetching policies makes it very difficult, not to say impossible, to design an interface provided by the cache allowing their integration. This is not a desirable feature as such an interface would be likely to reduce the cache efficiency. Instead, it would be better to alter the control flow and data flow of the cache on demand in order to integrate the prefetching policy. Nevertheless, web caches possess specific characteristics that motivate the need for a specific AOP infrastructure, providing the following features [22]:

Dynamic weaving and deweaving of aspects Policies running within a cache must change over time to cope with the characteristics of accessed servers.

Continuous servicing Loading or unloading a new policy must be done without losing the cache content. Additionally, service unavailability must be short enough to be masked by TCP/IP retransmission mechanisms.

Aspects for C++ programs Prefetching must be integrated within real Web caches such as Squid 3.0 that are written in C++.

Efficiency Policy execution must be as fast as possible to avoid degrading cache performance, both in terms of latency and bandwidth.

2.6 Summary

Prefetching and Caching are common techniques used in I/O systems to reduce latency. A Web cache acts as a relay between users and Web servers. It has three goals: reducing the user's latency, decreasing the consumed bandwidth and reducing the workload on the Web Server. But Web caches' benefit has limited by the rapid changes of objects in the Web. A solution is to anticipate future requests and prefetch the document in the cache [26]. Prefetching is complementary to the cache mechanism.

A prefetching policy results of the combination of two algorithm: a prediction algorithm and a threshold algorithm. The prediction algorithm is responsible of producing a list of the Web objects that are likely to be requested soon. Typical prediction algorithm use the user/server access history. The threshold algorithm is in charge of determining the number of Web objects to prefetch within this list depending on the resource available. Prefetching can be understood as a concern in Web caches. The bad news is that prefetching appears as a crosscutting concern with regards to typical Web cache structure. We believe that therefore Aspect-Orientation can help to implement prefetching within Web cache.

Chapter 3

State of the Art

Computer science evolved from assembly code to procedural programming, structured programming, functional programming and logic programming. Each step has improved the programmer ability to achieve a clear separation of concerns.

Currently, OOP (Object-Oriented Programming)is the most common programming paradigm. With OOP once builds software systems by decomposing a problem into objects. Objects abstract behavior and data into a single conceptional entity. Object-Orientation is reflected in the entire spectrum of current software development methodologies and tools. Nevertheless it has some limits: there are still concerns that fail to be separated clearly.

Aspect-Oriented Programming (AOP), is based on the idea that software systems would be better programmed by separately specifying the various concerns their relationship; then relying on mechanisms in the underlying AOP environment compose - weave - them together to build a program. AOP emerges as a new direction for software engineering. In this chapter, we first present the key concepts of AOP. The next section in this chapter levearages on this understanding of AOP key concepts to discuss how some programming languages have been extended with AOP constructs. This chapter concludes that there is no AOP extension allowing aspects to be woven on the fly in a C++ application.

3.1 Introduction to Aspect Oriented Programming

Thirty years ago, D. L. Parnas [19] already stated the importance of separation of concerns. *Concern* refers to the parts of software that are relevant to a particular concept, goal, or purpose. *Separation of Concerns (SoC)* refers to the ability to identify, encapsulate, and manipulate those concerns from the tangling codes. According to Kiczales et al. [9], a concern in an application should either appear in the source code as:

- A component if it is cleanly encapsulated in building block of the programming language.
- An aspect if it cannot be cleanly encapsulated in any construct of the programming language. Aspects are properties that crosscut components and tend to affect component performance and semantics.

Separation of concerns motivates AOP. In this section, we first describe this principle and its relationship to AOP. Then we will present the key concepts of AOP.

3.1.1 The separation of concerns principle

In a concrete complex system, there are a lot of concerns hidden in the implementation. These concerns can be divided into functional concerns and non-functional concerns. The former, functional concerns are circumscribed to the application domain. The typically functional concern deals with some basic functionality of the application domain. Non-functional concerns are not specific to the application domain and relates to housekeeping tasks. Typical non-functional concerns are dealing with real-time, persistence, distribution.

An Example - Logging Concern

Kiczales et al. [9] illustrates their approach with a bank system: a functional module, which copes with the business logic. It allows opening or closing an account, and withdrawing or depositing money. These tasks belong to the business logics of the bank system, so this module realizes a functional concern. A canonical example of a function concern is *logging*.

For the sake of security, the bank system has to record each manipulation on this system. In other words, a history of all the transactions has to be kept. Therefore, the logging concern takes place everywhere: in withdrawing, in depositing money by client and even for opening a new account.

With traditional programming paradigms, there is no way to clearly isolate the logging concern. It is mixed with the code of functional concerns. Therefore, all modules functions or methods involved in the different tasks (withdrawing, depositing or opening a new account) will contain at least a piece of code dealing with logging.

Crosscutting Concerns

When a modularized unit contains pieces of code dealing with different concerns, we say the code is *tangled*. If the code implementing a certain concern is contained in more than one modularized unit, That is *scattered* code. Tangled scattered codes are typical issues of traditional paradigms. They are revealing the necessity of a better mechanism enabling to separate different concerns. *crosscutting* concern is the one is scattered and tangled with the codes of other concerns. Crosscut is named from cutting across the several objects or methods.

3.1.2 Key concepts of AOP

Many AOP systems have been realized. But almost all of them share a common set of concepts. The most common structure used by AOP system is built on three concepts.

- **The base program** implements the logic of the application. concretely it implements all the concerns of the application that the preexisting language allow to separate cleanly. Thus the base program can be implemented using a traditional programming, non aspect-oriented paradigm.
- **Aspects** are implementing crosscutting concerns. These concerns cannot easily separated in the preexisting language. Aspects are isolating what would otherwise result in tangled or scattered code. They enable a better composition and reuse of the different concerns involved in the application.
- **The Weaver:** a mechanism is needed to compose the code of the non crosscutting concerns expressed in a traditional language with the crosscutting concerns. This composition named weaving is realized by a tool called weaver.

With aspect-orientation, the base program remains the same as in traditional programs but crosscutting concerns are implemented with aspects. In order to model and abstract crosscutting concerns, the aspect language should allow programmers to express how aspects should be composed with the base program. This mechanism relies in the following three elements.

- **A joinpoint** is a point in the base program that can be modified by an aspect. Joinpoints are well-defined points in a program's execution. It is a location in the base program where the control flow can be transferred to the aspect. Candidate joinpoints, for example, include calls to a method, a conditional check, a loop's beginning, or an assignment.
- **A pointcut** is a description of the execution contexts in which an aspect should be activated. Pointcuts are used to design joinpoints. Poincuts allows a programmer to specify where an aspect should be executed. They are expressed in a dedicated formalism: the pointcut language. They act as filters for the different joinpoints.
- **An advice** is the code implementing the functionality provided by an aspect. Joinpoint and pointcut define where and when control should be transferred to an aspect. Advices

are used to define the aspect behavior once it is executed. In other words, advices contain the code that will be run at the joinpoints designed by the pointcut. Additionally, advice should be able to retrieve information from the base program. This kind of special functionality should be supported by the AOP system. Generally speaking, the advice can be implemented in the same language that the base program. However, another implementation language can be used.

3.2 Implementation of AOP: An Overview

The main goal of an AOP system is to ensure that aspect and non-aspect code run together in a properly coordinated fashion. This coordination process is initialize at ***weaving*** time. It makes sure that the appropriate advices are run at the appropriate join points. This section discusses how to achieve weaving.

A given program P can always be viewed as a sequence of statements aimed at produce some result R. This result R is obtained through the execution of the program P. This execution is done by some platform (hardware, operating system, virtual machines etc.) that interprets the program's sequence of statements. Thus, any result R of a computation depends on both a program P and an interpreter I. A different result may be obtained by changing, at least, one of the elements of the couple $< P, I >$.

Bouraqadi-Saadani and Thomas Ledoux [3] propose to consider the set of the application components as a unique entity, which is a program P_0 written for some I_0. This couple $< P_0$,$I_0 >$ produces some result R_0. Aspects affect application execution to produce R_1. This new result can be obtained by changing, at least, one of the elements of the couple $< P_0$,$I_0 >$. The change is either a transformation of the program P_0, or a transformation of the interpreter I_0, or both transformations. These transformations allow to describe the whole set of possible approaches to implement a weaver.

3.2.1 Weaving through program transformation

This approach consists to transform only the program P_0. The interpreter I_0 is kept unchanged. A new program P_1 is built from both the application aspects and the program P_0. Each aspect is a set of transformation rules that refer to some join points. Aspect weaving consists to apply these transformation rules to the initial program P_0. This offers the advantage of leaving the interpreter I_0 untouched. This is especially interesting if the interpreter is not made in software but in silicium.

To use the pseudo-code program P_0 as an example

```
class A {
    void foo(){
        //some commands
    }
    void bar() {
        //some commands
    }
}
```

Then, if one wants to log in some file all invocations of `foo()` method, this logging can be an aspect to the joinpoint "invocation of the `foo()` method". Using the transformation rule, one can insert a log statement at the beginning of the `foo()` method:

```
Aspect Log{
    Insert 'logFile.write(currentDate)' in method foo();
}
```

To insert the log aspect in original program P_0 by aspect weaving. After weaving, the new program P_1 may look like:

```
class A{
    void foo(){
        LogFile.write(currentDate);
        //some commands
    }
    void bar(){
        //some commands
    }
}
```

3.2.2 Weaving through interpreter transformation

An alternative approach is to transform only the interpreter I_0 instead of P_0. A new interpreter I_1 is built from both the application aspects and the interpreter I_0. As a result, the interpretation P_0 is changed to take into account the different aspects.

```
Class Interpreter {
    Object invoke (Method m, Object[] args, Object receiver){
        //invoke Method m with arguments args on receiver
    }
    Object read (Field f, Object receiver){
        //return value of field f in receiver
    }
}
```

Then, one can define a log aspect as follows to log all method invocations in the original program P_0,

```
    Aspect Log{
        Insert 'if(m.name==foo()) logFile.write(currentDate)'
        in the method Interpreter::invoke()
    }
```

This aspect refers to the same joinpoint "invocation of the `foo()` method" in a different way - by interpreter's transformation. Then the new interpreter I1 might look like:

```
    Class Interpreter {
        Object invoke (Method m, Object[] args, Object receiver){
            If(m.name=="foo") logFile.write(currentDate)
        //invoke Method m with arguments args on receiver
        }
        Object read (Field f, Object receiver){
            //return value of field f in receiver
        }
    }
```

The rest of this section reviews the different aspect systems that have been devised. As it can be seen from table 3.1, most systems extends a language, usually Java. Because our prfetching policies should be deployed on demand we pay a particular attention to the weaving time. The composition of the aspects and the base program at weaving time, can be done on demand, that is dynamically at runtime or statically at compile time.

3.2.3 Static weaving

Static weaving [6] is usually achieved by modifying the the source code of a class to aspect specific statements at join points. In other words, the aspect code is inlined into the base program. The result is a highly optimized woven code whose execution speed is comparable to code written without AOP.

	Weaving Transformations	Weaving Type	Language
AspectJ	Program Transformation	Static	Java
AspectC	Program Transformation	Static	C
AspectC++	Program Transformation	Static	C++
TinyC^2	Program Transformation	Dynamic	C
JAC	Program Transformation	Dynamic	Java
PROSE	Interpreter Transformation	Dynamic	Java
LOOM.NET	Program Transformation	Dynamic	C#
μDyner	Program Transformation	Dynamic	C#

Table 3.1: Comparison of Aspect-Oriented Programming tools

AspectJ: An AOP implementation for Java

AspectJ is a freely available AOP implementation for Java from Xerox PARC. AspectJ design's goal is to be a compatible extension to Java [13] in order to ease its adoption by current programmers. AspectJ provides:

- **Upward compatibility** all legal Java programs are legal AspectJ programs.
- **Platform compatibility** all legal AspectJ programs run on standard Java virtual machines. AspectJ's weaver takes the form of an aspect compiler. Because the final code generated by the AspectJ compiler is expressed in standard Java byte code, it can run on any standard Java Virtual Machine (JVM).

AspectJ extends Java with support for two kinds of crosscutting implementation [13]. It first makes it possible to define additional implementation to run at certain well-defined points in the execution of the program. Secondly, AspectJ enables to define new operations on existing types. We call this static crosscutting because it affects the static type signature of the program. In addition, AspectJ also allows the "aspecting" of other aspects and classes in several ways. Indeed, you can introduce new data members and new methods, as well as declare a class to implement additional base classes and interfaces.

AspectC

AspectC is a simple aspect-oriented functional language. The initial design of AspectC was taken directly from the non-object oriented subset of AspectJ. The language has them evolved to acknowledge the differences between C and the non-OO subset of Java. Aspect code, known as `advice`, interacts with primary functionality at function call boundaries and can run `before`, `after` or `around` the call. The central elements of the language are a means for designating particular function calls, for accessing parameters of those calls, and for attaching advice to those calls.

AspectC++

AspectC++ is a general-purpose aspect-oriented extension of C++ [10]. Most of the basic constructs of AspectC++ are modelled from AspectJ. AspectC++ is implemented as a C++ preprocessor; based on PUMA [10]. PUMA is a source code transformation system for C++. This architecture is shown in Figure 3.1. First the AspectC++ source code is scanned, parsed and a semantical analysis is performed. Then, at the planning stage, the pointcut expressions are evaluated and the join point sets are calculated. A plan for the weaver is created containing join points and the operations to be performed at the join points (i.e. adding advice code). While the planing stage is mainly independent from C++, the weaver is responsible to transform the plan into concrete manipulation commands based on the C++ syntax tree generated by the PUMA parser. The actual code manipulation is then performed by the PUMA manipulation engine. The output of the prototype compiler is in C++ source code with the aspect code woven in. The produced output source code does not contain AspectC++ language constructs anymore and thus can be translated to executable code using conventional C++ compilers.

3.2.4 Dynamic weaving

The compilation of the base program and the aspect code: weaving can be done dynamically or statically. For example, the weaving operation can be understood as a scheduling operation that will yield the execution of the aspect at appropriate times.

TinyC^2: a dynamic weaving aspect language for C

The design goal of the TinyC^2 language [27] is to provide a language perspective in terms of code instrumentation, and, at the same time, to establish a framework for implementing a post-compilation weaving aspect language that uses the C syntax. It uses a third party instrumentation tool as the back-end. Similar to AspectJ using standard Java grammar, TinyC^2 uses the standard C grammar extended with a few new syntactic constructs. Programmer can use the regular C syntax to compose code blocks. However, the basic modularization units in TinyC^2 are not functions but ”snippet”. A snippet is a unit of aspect implementation. It encapsulates a code block and defines the ”weaving” points in the component program where the aspect code is inserted. Snippets are functionally equivalent to the ”joinpoint” and ”advice” concepts in an ”aspect” module in AspectJ.

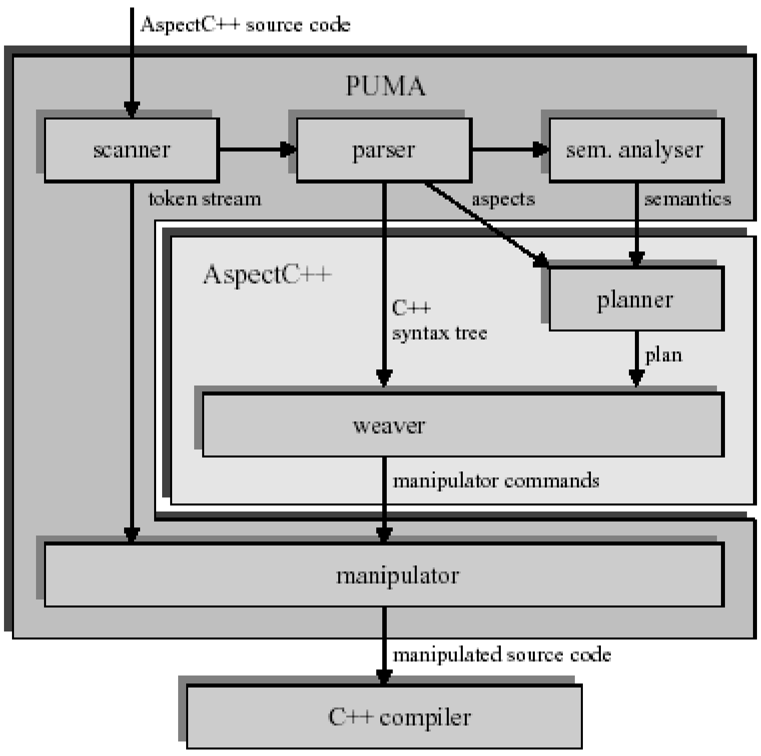

Figure 3.1: Architecture of the AspectC++ compiler

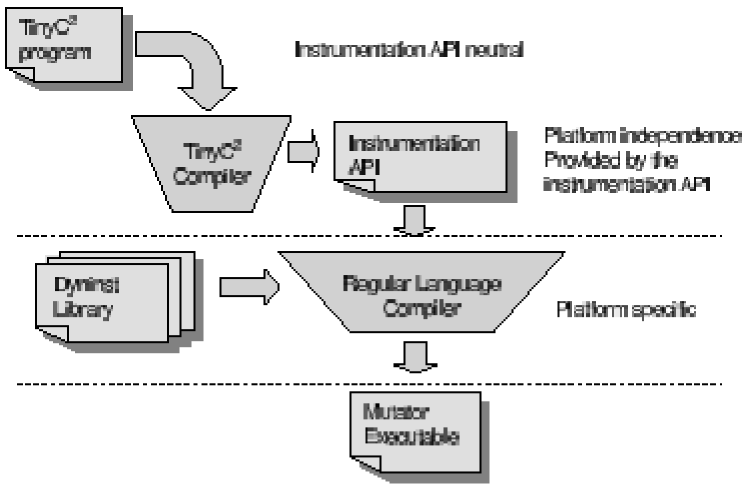
TinyC² program
Instrumentation API neutral
TinyC² Compiler
Instrumentation API
Platform independence Provided by the instrumentation API
Dyninst Library
Regular Language Compiler
Platform specific
Mutator Executable

Figure 3.2: Compilation process of TinyC^2

TinyC^2's compiler [27] perform a source-to-source translation using the ANTLR parser generator tool. It consists of three main components: the grammar file for the language, the lexer and parser generated form the grammar file, and the back end code translator and generator. The most fundamental component of the translator is the Snippet class which is the abstraction of the generated code for a particular language element in TinyC^2. Programs written in Tiny^2 language are translated by TinyC^2 compiler to a source file written according to the application programming interface of the target instrumentation platform. The generated source file then can be compiled again using the common language compiler of the runtime platform. It is the responsibility of the instrumentation platform to integrate the generated aspect system and the component program together. This process is illustrated on Figure 3.2.

The current implementation of the back end code generator is targeted at the Dyninst runtime instrumentation platform. Therefore, the TinyC^2 code is firstly translated into C code calling the Dyninst library API [27]. The translated code is then compiled by a regular C compiler to generate a binary executable which is linked to the Dyninst instrumentation library. The executable is started with the process information of the target running system. The Dyninst library is responsible for properly inserting the code into the address space of the target program. Thus, TinyC^2 can be classified as a dynamic weaving aspect language.

Java Aspect Component (JAC)

JAC (Java Aspect Components) [23] is a framework for aspect-oriented programming in Java, not a new language. JAC is object-based and does not require any syntactic extension to Java. It uses the Javassist class load-time MOP. In JAC, an aspect is a regular Java object. An aspect

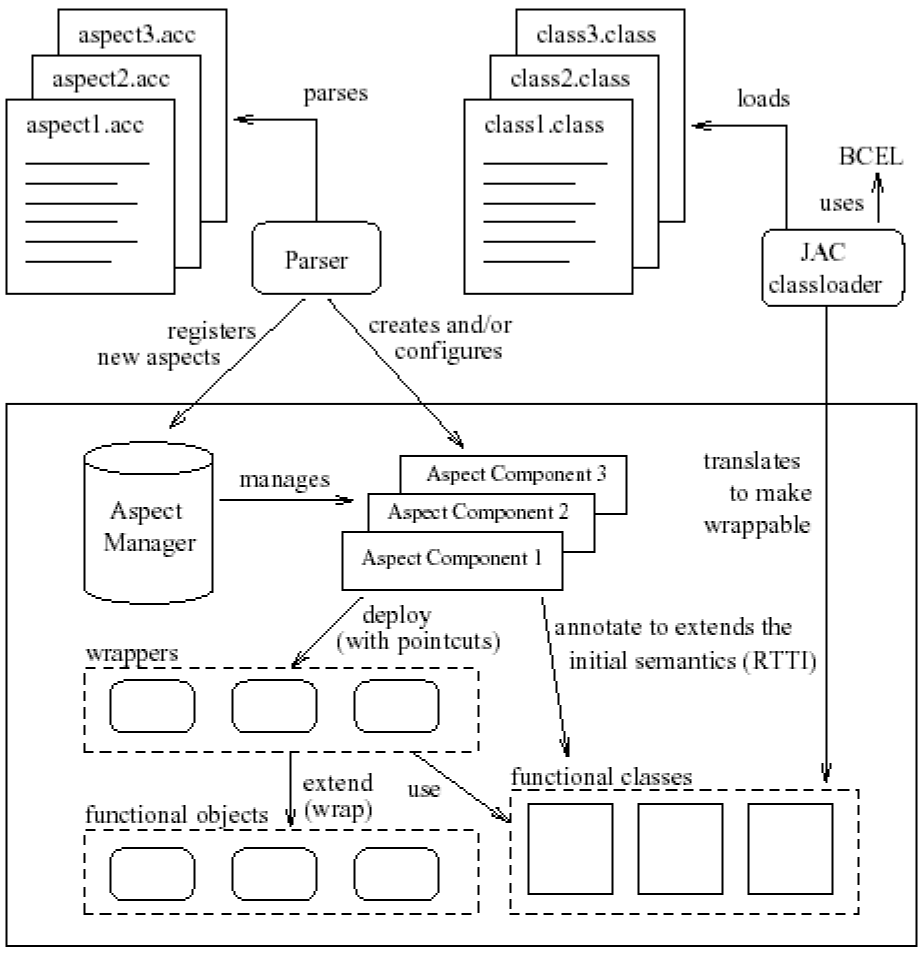

Figure 3.3: The JAC architecture

program in JAC is a set of aspect objects that can be dynamically deployed and undeployed on top of running application. Aspect objects may define three kinds of aspect methods wrapping methods (that wrap application methods and provide the ability to run code before and after the wrapped methods); role methods (that add new functionalities to application objects) and exception handlers. The aspects composition issue is handled through a well-defined wrapping controller that specifies for each wrapped object at wrap time, runtime or both, the execution order of aspect objects.

Figure 3.3 shows how the different JAC system objects interact with the application objects to implement the aspects semantics. On the right side of the figure, one can see that the functional classes are modified at the bytecode-level in order to make their instances wrappable. The JAC MOP implementation uses a load-time transformative technique that inserts hooks towards the wrappers. These hooks are reflective invocations so that the actual wrapping method can be resolved at runtime. Once the classes are translated, they are ready to be wrapped by the aspect components. When an aspect is woven to a given application, the JAC system first reads the available aspect component configuration (*.acc files on the upper left of the figure). The parser then invokes a set of configuration methods on the newly instantiated aspect component. These invocations trigger the creation of pointcuts and the tagging of the functional classes with some meta-data (through the RTTI). When a new instance is created, the AC-Manager (Aspect-Component-Manager) automatically notifies all the registered aspects so that the pointcuts wrap its methods according to the aspect configuration. The important point here is that any aspect component can be woven or unwoven at any time, moreover, its configuration file can be parsed again while the application is running. When an aspect is unwoven or when a new configuration is read, all the added meta- data and the pointcuts are destroyed by the system. This is possible thanks to the wrapping mechanism implementation for dynamic wrapping/unwrapping.

JAC wrappers can be added or removed at runtime. Contrary to regular wrappers, which delegate to the wrappee and implement the same interface as the wrappee in most of the time, dynamic wrappers rely on a Meta-Object Protocol that uses reflection for its implementation. The JAC Meta-Object Protocol uses a load-time tranformative technique that inserts hook towards the wrappers. These hooks are reflective invocations so that the actual wrapping method can be resolved at runtime.

PROgrammable extenSions of services (PROSE)

PROSE (PROgrammable extenSions of sErvices) [20] is a platform based on Java. It supports dynamic weaving. Aspects are expressed in the same source language as the application (Java), and PROSE allows aspects to be woven, unwoven, or replaced at run-time, providing for speeding up the design-test cycle. In PROSE, aspects are written in Java. No separate tools are needed: PROSE is based on the debugger interface of the virtual machine. By expressing aspects in the source language, PROSE allows the definition of customized AOP constructs.

The goal of PROSE is to allow the definition of aspects following existing AOP solutions, while avoiding the complexity of Meta Object Protocols. In PROSE, aspects are defined as classes and aspect are objects. An aspect contains the information about the matched join-points as well as the advice action and may be stateful like any other object. The AspectJ counterpart of the PROSE aspect uses the same key words: **before**, **after** to define how the crosscut is woven into the original code.

The PROSE uses the run-time approaches to implement AOP [20]. Run-time application changes can be achieved using meta-object protocols (MOPs), which have been used to express crosscutting concerns before the notion of AOP was proposed. A straightforward run-time approach to AOP for Java is to locate the support for weaving and unweaving aspects directly in the Java Virtual Machine (JVM). JVM also provide an interface for weaving aspects at run-time, called Java Virtual Machine Aspect Interface (JVMAI). The PROSE JVMAI is designed as a JVM plug-in.

AOP with C# and .NET (LOOM.NET)

"Dynamic aspect weaving" [21] means the at a component (a target class) and an aspect class will become woven during runtime. In LOOM.NET, an aspect is a simple C# class derived from the base class `Aspect`. An aspect class works only in conjunction with an instance of another class. At a *connection point* both will become woven. Methods of the aspect class can be identified as connection points, which is indicated by the C# **call** attribute above the method definition in the aspect class. During dynamic aspect weaving, all of the connection points inside an aspect class will become woven with a target class's method if some requirements is met [21]. The context property of the aspect base class allows access to an object of type `AspectContext`, which contains the required information. Schult et al. [21] gave a solution based on Microsoft .NET and implemented for dynamic aspect weaving in a .NET library. This library provides several classes and attributes defined within the namespace Aspects:

Aspect is the base class for all defined aspects.

Weaver is a class which implements the weaving functionality.

Call is an attribute to define connection points.

AspectContext allows invocation of instance methods via `Aspect.Instance`.

As described above, the Aspects namespace contains a class called `Weaver`. It provides a function named `CreateInstance` to weave a given target class. This function does the same as the new statement - it creates a new object of a given class (the target class). This function does the same as the new statement - it creates a new object of a given class (the target class). But furthermore this function weaves the target class with an aspect-object. This can be done in two ways: dynamically and staticallly. The dynamic version is as follows:

```
A a=Weaver.CreateInstance(typeof(A), null, new MyAspect()) as A;
```

Giving the aspect instance explicitly as a parameter to `CreateInstance` is more flexible than naming it via attribute - as the aspect and its parameters can be identified at runtime. The code implementing dynamic aspect weaving first looks for a custom attribute derived from Aspect. If there is no aspect given, the `CreateInstance` call is equivalent to new `A(args)`. What happened during the creation is illustrated in Figure 3.4.The weaver looks for connection points and tries to join them with the target class's methods.

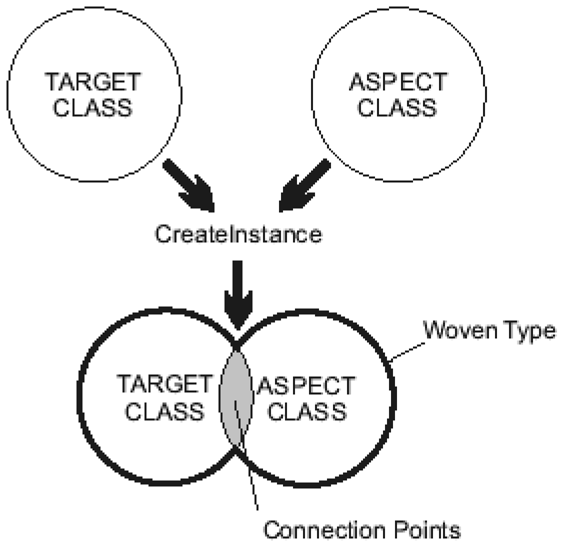

Figure 3.4: The weaving process

micro DYNamic weavER (μDyner)

The μDyner Aspect-Oriented Programming infrastructure dynamically deploys aspects in running C programs. Generally speaking, μDyner realizes the weaving process by directly rewriting the code being executed. Figure 3.5 represents the architecture of μDyner.

From an architecture viewpoint, μDyner can be divided into two parts. The first one deals with the aspect generation: it is essentially based on an aspect compiler. The second part is in charge of weaving the compiled aspects into the base program: this part relays essentially on a kernel library loaded with the base program.

μDyner uses standard library mechanisms to force the kernel library to be loaded by the base program. A kernel library: `uDDT.so` receives and handles the weaving requests. Weaving requests allows third part processes to deploy compiled aspects in the base program. Compiled aspects are produced by an aspect compiler provided by μDyner. This compiler first translates the aspect code into C and relays on a regular C compiler (`gcc`) to generate the executable code. As shown on Figure 3.5, compiled aspects are packaged as shared libraries.

We use a toy example: `TestBase.c` for the sole purpose of illustration. The usage of μDyner can be roughly separated into three steps: configuring the system, compiling the aspects and weaving.

Configuration of μDyner

μDyner is a flexible system: it delays as much as possible the binding between the components involved in its architecture. In order to achieve this goal, a few shell variables are used to locate the different components. Therefore, a user should first set these variables to allow proper operations of the system.

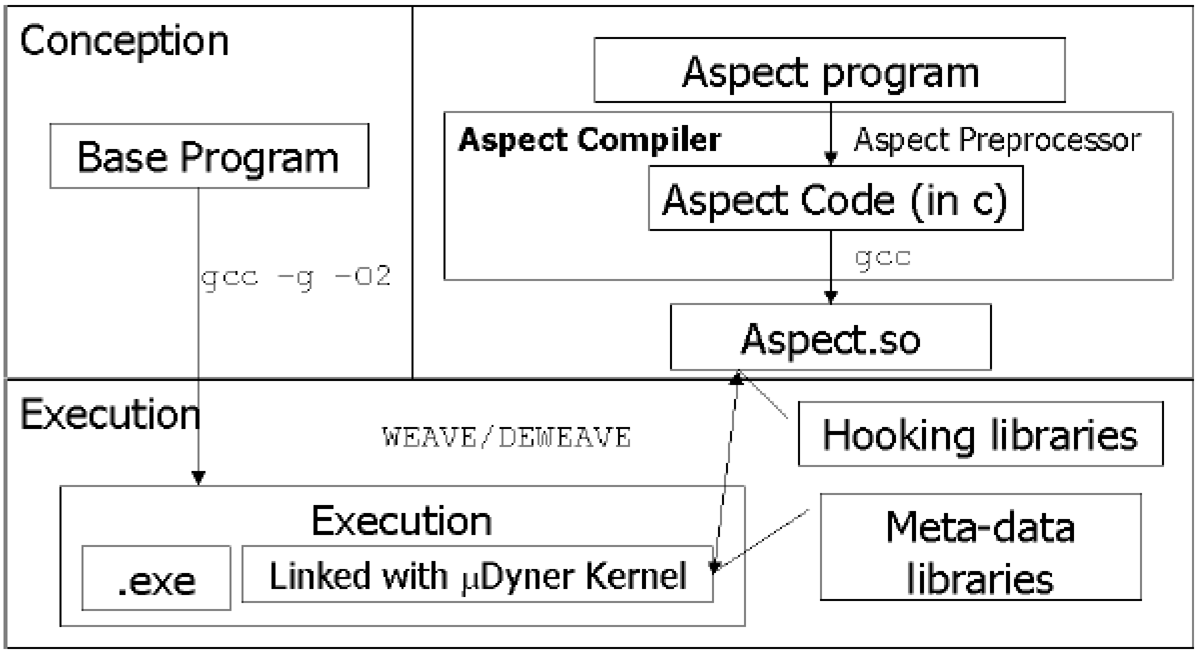

Figure 3.5: The architecture of μDyner

First of all, a shell variable is used to locate the directory where the μDyner system is installed. Using a bourne shell, it can be set:

```
export UDDT_PATH=/home/ychen/uDDT
```

The kernel is a shared library dynamically linked with the base program through the `LD_PRELOAD` shell variable. Using a bourne shell again:

```
export LD_PRELOAD =
/home/ychen/uDDT/Base-Program-ToolBox/Support/libuDDT.so
```

The aspect compilation chain in μDyner is open. This openness is achieved by letting advanced users specify how the aspect code should be translated in C code. Less advanced users control this process by setting the shell variable `UDDT_ASPECT_PREPROCESSOR`. For example, Again with a bourne shell:

```
export UDDT_ASPECT_PREPROCESSOR =
/home/ychen/uDDT/RewritingSite/ReadGlobalVariable/AspectPreprocessor
/preprocessor
```

In order to ease aspect development, one can set the shared library-searching path with:

```
export LD_LIBRARY_PATH=.
```

Compilation of the base program

The current implementation of μDyner involves rewriting on the fly the object code of the base program. It exploits debugging information to determine the rewriting sites. The analysis performed by μDyner do not currently support function inlining. Therefore the base program should be a C program compiled with debugging information and without function inlining. With `gcc`, the base program should have been compiled with `-g` and with a level of optimization not high than `-O2`. Outside these two restrictions, no special modification of the base program is required.

Compilation of the aspect

The aspect `Aspect.c` can be compiled in `Aspect.so` as follows:

```
../uDDT/Preprocessor/AspectPreprocessor/preprocessor Aspect.so
Aspect.c
```

Weaving

Before weaving, the base program is executed:

```
./TestBase
```

Then, we weave the aspect into the base program:

```
../uDDT/Weave/weave 29136 WEAVE Aspect.so
```

(`29136` is the process id of example `TestBase` in our experiment.)

Hereto, the aspect is successfully woven into the base program by μDyner aspect system.

Aspect Syntax

The syntax of the aspect language is defined by [22] in Figure 3.6

```
<aspect>::=<name>":["<filters-advice>"]"
<name>::=<identifier>
<filter-advice>::=<function-invocation>"["<filters-advice>"]"
                       |<function-invocation>"["<advice>"]"
                       |< global-variable-access>"["<advice>"]"
<function-invocation>::=<type><identifier>"("<params>")"
<type>::=<C-type>
<params>::=<type><identifier>|<params>","<params>
<global-variable-access>::=<global-variable-read>
                          |<global-variable-write>
<global-variable-access>::=<type><identifier>
<global-variable-access>::=<type><identifier>"=" <identifier>
<advice>::=<C-compond-instruction>
```

Figure 3.6: The aspect language syntax

3.3 Summary

Aspect-Oriented Programming (AOP) is an emerging programming paradigm and philosophy. Its origin goes back to several approaches focusing on Separation of Concerns (SoC). AOP is based on three elements: joinpoint, pointcut, advice. Most AOP languages today, including AspectJ, Hyper/J [1], AspectC and AspectC++ statically transforms program either at the source-code level or at the byte code level. The transformation of the AspctC# (LOOM.NET), TinyC^2 and μDyenr are done dynamically. PROSE uses the interpreter transformation. The Meta-Object Protocols (MOPs) has been used to express crosscutting concerns by JAC. Last, we draw a conclusion: there is no AOP tool supporting dynamically weaving aspect in C++ application.

Chapter 4

Contribution

Our approach is motivated by the need to integrate on demand prefetching policies into Web caches. Aspect-Oriented Programming is a solution, as explained in the previous chapter. However no dynamic weaver is available for C++. AspectC++ [10], for instance, allows only to weave aspects at compile-time. On the other hand, μDyner provides the ability to weave aspect on the fly but only in C applications. In this chapter, we identify the problems preventing μDyner to be applied to C++ applications.

The syntax of the μDyner aspect language is defined as an aspect name followed by a nested sequence of pointcuts before a block of C code containing the advice. First of all, the pointcuts offered by the extension of μDyner for C++ should be C++ specific, in other words, different from these provided by μDyner for C. Understanding the source level differences between C and C++ is consequently crucial to determine the appropriate set of pointcut that the aspect language should provide.

C and C++ are compiled by different compilers and the code generated, despite the fact that C++ is a superset of C, is different. The differences between a compiled C program and a compiled C++ program prevents μDyner weaving mechanism to be used directly on a C++ application. Therefore it is primordial to understand the difference between C and C++ code at the binary level to design an appropriate weaving mechanism.

The rest of this chapter is structured as follows: it studies the difference between C and C++ at the source level and at the binary level. Next, this chapter describes the solutions we propose allowing to dynamically weave aspect in C++ programs.

4.1 Source level difference between C and C++

C++ can be understood as a superset of C. Almost all of the features and constructs available in C are also available in C++. C++ is object-oriented. It therefore supports polymorphism thanks to function overriding and function overloading and provides multiple inheritance.

4.1.1 Function Overloading

C++ supports function overloading. In other words, the same function name can be defined more than once with different formal parameters. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler determines which definition is intended to be used by the context in which the function occurs. In other words, the compiler picks the appropriate version of the function based on the arguments with which it is called.

Syntactically, if one wants to overload a function called `ave()` computing the average of two numbers:

```
//Find the average of two integers
float ave (int a, int b) {
     return float (a+b)/2.0;
}
```

and

```
//Find the average of an array of n integers
float ave (int a[], int n) {
     int sum = 0;
     for (int i = 0; i < n; i++)
          sum += a[i];
     return float (sum) / n;
}
```

The function `ave()` is overloaded. It can be used:

```
    int a = 2, b = 3;
    int x[5] = {10, 20, 30, 40, 50};
    cout << "The average of 2 and 3 is " << ave (a, b) << endl;
    cout << "The average of the integers 10, 20, 30, 40,"
         << " 50 is" << ave (x, 5) << endl;
```

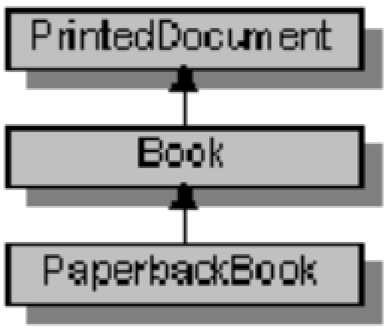

Figure 4.1: Single inheritance graph

Thus the pointcut language of an aspect system for C++ should allow to distinguish the different version of an overloaded function.

4.1.2 Inheritance

Syntactically, considering the inheritance tree from Figure **??**, yields the following C++ class hierarchy:

```
class PrintedDocument{
    //Member List.
};

//Book is derived from PrintedDocument
class Book: public PrintedDocument{
    //Member List.
};

//PaperbackBook is derived from Book
class PaperbackBook: public Book{
    //Member List.
};
```

In an inheritance tree, the derived class contains the members of the base class plus zero or more new members. As a result, a derived class can refer to members of the base class (unless those members are redefined in the derived class). The scope-resolution operator (`::`) can be used to refer to the members of the base classes when those members have been redefined in the derived class.

```
class Document{
    public:
        char *Name;
        void PrintNameOf();
};

void Document::PrintNameOf(){
    cout<<Name<<endl;
}

class Book: public Document{
    public:
        Book(char *name, long pagecount);
    private:
        PageCount;
};

Book::Book(char *name, long pagecount){
    Name = new char[strlen(name)+1];
    strcpy(Name,name);
    PageCount = pagecount;
}
```

The pointcut language of an aspect system for C++ should therefore take the scope into account.

4.1.3 Function Overriding

The redefinition of a virtual function in a derived class is called *overriding*. Syntactically, the declaration of the overridden functions are marked with a keyword `virtual`. If a function is declared as `virtual` in the base class, it is `virtual` in all the derived classes. All derived-class functions that match the signature of the base-class declaration will be called using the `virtual` mechanism. At compile time, the compiler does not know the addresses of a virtual function call. Therefore, the virtual mechanism employs a scheme named late binding that allows to delay the function lookup at runtime. By contrast, C function can be invoked only using early binding.

The following example highlights the C++ syntax. It shows a base class providing an implementation of the `PrintBalance` function:

```
class Account {
    public:
        Account( double d );          // Constructor.
        virtual double GetBalance();  // Obtain balance.
        virtual void PrintBalance();  // Default implementation.
    private:
        double _balance;
};

//Implementation of constructor for Account.
double Account::Account( double d ) {
    _balance = d;
}

//Implementation of GetBalance for Account.
double Account::GetBalance() {
    return _balance;
}

// Default implementation of PrintBalance.
void Account::PrintBalance() {
    cerr << "Error. Balance not available for base type."
         << endl;
}
```

Two derived classes, `CheckingAccount` and `SavingsAccount`, can be created as follows:

```
class CheckingAccount: public Account{
    public:
        void PrintBalance();
};

//Implementation of PrintBalance for CheckingAccount.
void CheckingAccount::PrintBalance() {
    cout << "Checking account balance: " << GetBalance();
}

class SavingsAccount: public Account{
    public:
        void PrintBalance();
};
```

```
//Implementation of PrintBalance for SavingsAccount.
void SavingsAccount::PrintBalance() {
    cout << "Savings account balance: " << GetBalance();
}
```

The `PrintBalance` function in the derived classes is `virtual` because it is declared as `virtual` in the base class, Account. To call virtual functions such as `PrintBalance`, code such as the following can be used:

```
// Create objects of type CheckingAccount and SavingsAccount.
CheckingAccount *pChecking = new CheckingAccount( 100.00 );
SavingsAccount  *pSavings  = new SavingsAccount( 1000.00 );

// Call PrintBalance using a pointer to Account.
Account *pAccount = pChecking;
pAccount->PrintBalance();

// Call PrintBalance using a pointer to Account.
pAccount = pSavings;
pAccount->PrintBalance();
```

If a class is declared that does not provide an overriding implementation of the `PrintBalance` function, the default implementation from the base class `Account` is used.

Functions in derived classes override `virtual` functions in base classes only if their types is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.

Because calls to virtual functions are dispatched in a different manner than C style function, the pointcut language of an aspect system for C++ should consider the virtual function.

4.1.4 Multiple Inheritance

As shown in Figure 4.2, in C++, a given class can inherit from more than one super class. This property named: "multiple inheritance" may introduce naming conflicts in derived class.

Multiple inheritance improves the programmers ability to combine classes. To be fully exploited multiple inheritance is normally used in conjunction with virtual functions. Syntactically, one can declare `CollectibleString`, derived from `Collectible` and `String`, as following:

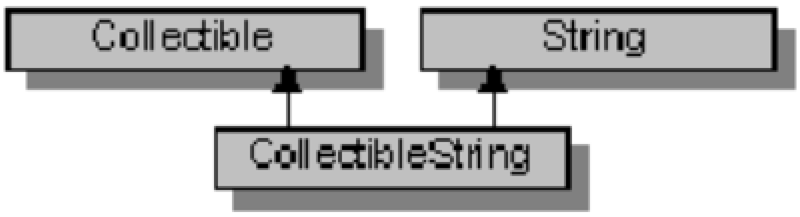

Figure 4.2: Multiple inheritance graph

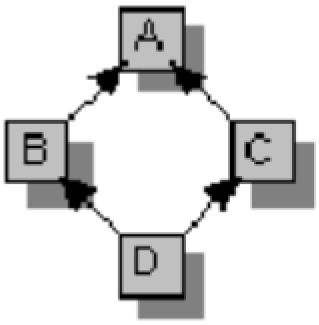

Figure 4.3: Virtual inheritance graph

```
class CollectibleString: public Collectible, public String{
    //new members;
};
```

Multiple inheritance raises however specific problems. Consider, for example, the class hierarchy in Figure 4.3. `A` is the base class for both `B` and `C`. However, when both classes are combined to form `D`, the newly formed class `D` will contain two subobjects of type `A`, one from `B` and the other from `C`. To solve the problem C++ provides a mechanism called virtual inheritance.

The following C++ code declares `A` to be a virtual base class:

```
class A{
    //member list
};

class B: public virtual A{
    //member list
};
```

```
class C: public virtual A{
    //member list
};

class D: public B, public C{
    //member list
};
```

The virtual inheritance mechanism ensures that D will contain only one A object. Thus virtual inheritance complicates the inheritance problem that the pointcut language should cope with.

4.2 Binary level difference between C and C++

The previous subsection has described the main characteristics of C++ source code. In order to extend μDyner for C++, the original pointcut language should be modified to reflect the way function calls are dispatched in C++. But before thinking to reshape the pointcut language, a weaving mechanism for C++ is needed. The design of this mechanism requires to understand the differences at the binary level between C and C++ preventing its original μDyner weaving mechanism to be applied in C++ application. This understanding is the purpose of this section.

4.2.1 Name mangling

At the binary level after compilation, functions are designated by symbols. A symbol is a unique name within an executable file. In C++ symbols names are produced at compile time by decorating the original function or variable name.

For example, member names are made unique by concatenating the name of the member with that of the class. For example, given the declaration

```
class Bar {
    public:
        int ival();
        ...
};
```

The symbol used by `g++` for `ival()` looks like:

```
// a possible function name-mangling
ival_3Bar()
```

This scheme known as "name mangling" allows the source code to provide overriding and overloading while using still at the binary level code to use unique symbol names to refer to variables and functions.

4.2.2 Virtual Member Functions

In the following `class1`, the method `vfunc1()`, `vfunc2()` and `vfunc3()` are all declared as virtual functions. The compiler will connect them to their function body at runtime due to the late binding mechanism.

```
class Class1{
    public:
        m_data1;
        m_data2;
        virtual vfunc1();
        virtual vfunc2();
        virtual vfunc3();
};
```

`g++` uses vprt and vtable to support the virtual function. A table of pointers to the different virtual functions in generated for each class. This table is called the virtual table (vtable). A single pointer to the associated virtual table is inserted within each class object. This pointer is called the vptr. Vtable and vptr are generated and manipulated behind the scenes by the C++ compile and the C++ runtime system.

Each time one creates a class containing virtual functions, or one derives a class from a class containing virtual functions, the compiler creates a unique vtable for that class. Three cases are occur:

- A new virtual function not present in the base class can be introduced. In this case, the virtual table is grown by a slot and the address of the function is placed within that slot.
- A newly introduced function overloads a function of the base class. It inherits the virtual function declared within the base class. Literally, the address of that virtual function is copied into the associated slot in the derived class's virtual table.
- A newly introduced function overrides another function. In this case, the virtual table is grown by a slot and the address of the function is placed within that slot.

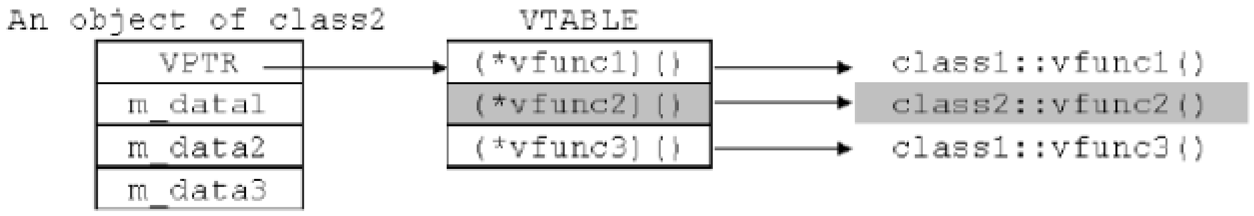

Figure 4.4: Vtable layout of class2

There is only one vptr for each object with simple inheritance. This vptr is statically initialized to point to the appropriate vtable. For example, we have another class `class2`, derived from `class1`.

```
class Class2: public class1{
    public:
        m_data3;
        virtual vfunc2();
};
```

In this case, in the vtable of `class2`, the compiler places the address of all the functions that are declared virtual in this class or in the base class. Because the `vfunc2()` is overridden, the compiler will associate the pointer referred to `class1::vfunc2()` with the new address for `class2::vfunc2()`. `vfunc1()` and `vfunc3()` have not been overridden, the compiler uses the address of the base-class version in the derived class for the virtual functions. The memory layout of `class2` is expressed on the Figure 4.4.

Multiple inheritance and virtual inheritance also use virtual function to achieve their late binding.

There are more pointers involved in vtable under multiple inheritance and virtual inheritance, which is caused by relationship of the derived class with its second and subsequent base class subobjects.

4.3 A dynamic weaver for C++

The two previous sections have established that a number of issues arise preventing to apply μDyner to C++. These issues can be roughly classified in language level issues and binary level problems. Language issues reveal that a pointcut language originally built for C programs would be inadequate for C++. The problem can be summarized in one sentence: C and C++

```
    aspect-library ::= aspect | aspect aspect-library+
                                | C++-compound-instruction aspect-library+
            aspect ::= aspect-name "[:"pointcut-advice":]"
  pointcut-advice ::= function-call "[:"pointcut-advice ":]"
                                | function-call "[:" advice":]"
                                | global-var-access
     function-call ::= type identifier-or-star "("params ")"
                                | type identifier-or-star "()"
            params ::= type identifier | params ","params
 global-var-access ::= global-var-read | global-var-write
   global-var-read ::= type identifier-or-star
  global-var-write ::= type identifier-or-star "=" identifier
            advice ::= C++-compound-instruction
identifier-or-star ::= identifier | "*"
       aspect-name ::= identifier
```

Figure 4.5: An aspect language for C++

uses different mechanisms to identify variables and functions. In C, a variable or a function is designated by a name, unique in the translation unit. In C++, a variable or a function is designated by its name and the class it belongs to and plus for functions, the arguments list.

The binary issues shows that the rewriting techniques provided by μDyner can not be applied as if to C++. These issues prevent the original μDyner weaving mechanism to work on C++ application. At the binary level, the problem is twofold. First of all, μDyner made the assumptions that the symbol names corresponds to the names appearing in the aspect pointcut source code. As it has been explained in section 4.2.1, this is not the case in C++. C++ employs a name mangling algorithm. Secondly, μDyner was designed for C, a language that provides only early binding. Instead, in order to support object-oriented feature, C++ uses early and late binding. C++ uses vtable to implement late binding. In the following subsections, solutions to these problems are presented.

4.3.1 Language level issues

We propose to keep almost the same pointcut language than μDyner. C-type becomes C++ types. C-compound instructions becomes C++-compound instructions. These choices solve the language issues. The BNF we proposed is described in figure 4.5. It remains simple and can be understood by programmers as an extension of the μDyner language for C.

Contrary to most aspects language that extends an object-oriented base language, aspects are not objects in our approach. The relationships between objects and aspects is still unclear. Among different issues, one can wonder what would mean the inheritance relationship between aspects or how what will be the sense of a constructor method for aspects. One can also wonder if aspects belong to the meta level part of the object-model rather than the base level. Our proposition eludes these problems while allowing different approaches to be experienced. For example, a programmer can choose that the advices of all the aspects will delegate their work to an instance method of some C++ class. Templates and macros can even be used to automate this kind of approach. The advantage of our proposition is that an aspect does not have to be an object if it is not needed. Thus, with our aspect language aspects can be extremely lightweight and still allow programmers to experiment different relationships between aspects and objects.

4.3.2 Binary level issues

Binary level issues prevent the efficient weaving techniques proposed by μDyner to be applied on C++ program. This is very unfortunate since we believe that efficient aspect systems will ease the acceptance of AOP in the industry. In this subsection, we show how to devise an efficient weaving scheme for C++ by extending the μDyner proposition.

Symbols

μDyner uses `nm` to translate the names defined in the source code into symbol names in the object file. `nm` is a command the posix tool that extracts the symbols contained in compiled program. Invoked with proper options, `nm` will demangle the symbol names contained in the executable file. Still deeper modifications of μDyner are required. For example, μDyner parsing algorithms need to be modified to take into account the type of the arguments of the function and not to limit themselves to the function name only. To implement our extension, we have modified μDyner to invoke properly `nm` as well as its parsing algorithm.

`nm` involved with the appropriate option can demangle C++ symbol.

Late binding

μDyner analyzes the object code of the application. It scans the object code looking for function calls or variable accesses. These calls or accesses are detected when the proper instruction refers the address of the function body or of the variable. But this approach is not appropriate for C++ virtual functions. The object code lookups the address of the functions using the vtable. Thus, the address of `virtual` function is not hard coded in the object code as it is for early bound function. μDyner proposition is to rewrite the call site of the functions. We propose to keep this scheme for early bound C++ functions and for virtual functions to alter the vtable.

When the pointcut of an aspect ends with a function call declared as `virtual` in the base program, we propose to change the address that will be provided by the vtable to a new function. This new function will be responsible of checking the remaining elements of the pointcuts. If necessary, it will execute the advice of the aspect. Otherwise, it will call the function originally provided by the vtable. This new function should be generated behind scenes by the aspect compiler.

As explained before, μDyner uses `nm` to the retrieve symbols from a compiled program. `nm` unfortunately only provides the address and the size of the vtables available in the compiled program. In other words, `nm` does not describe what the different vtables are containing. In order to be able to replace the appropriate pointer in the vtable, such a knowledge is nevertheless necessary.

If `nm` describes the content of the different vtables, every vtable function has an associated symbol that `nm` retrieves. When appropriate, the address of this symbol is contained in a slot of vtable.

Therefore, we first scan all the function symbols retrieved by `nm`. Then we scan each slot of each vtable and compare it with the addresses of the different functions. We wrote a postprocessor that stores the content of the vtables in a meta-data-file. At weaving time, this meta-data allows our extension of μDyner to exchange the appropriate pointer in the vtable with almost any lookup overhead.

4.4 Summary

Being motivated by the dynamic integration prefetching policies into Web caches, we extended μDyner for C++ because the original one provided to dynamically weave aspect in running C application. This chapter firstly discussed the difference between C and C++ at the source level and binary level. Based on those difference, we proposed our proposition. At the language level, we kept almost the same pointcut language than the original μDyner by changing C-compound instructions into C++-compound instructions. At the binary level, We proposed to invoke a proper option of `nm` for demangling the symbol names contained in the executable file. We proposed to keep the original scheme for early bound C++ functions. But for the pointcut of an aspect ending with a virtual function, we proposed to change the address that will be provided by the vtable to a new function.

Chapter 5

Implementation

In chapter 3, an overview of μDyner (micro DYNamic weavER) has been given. This tool allows to dynamically weave and deweave aspects in running C applications. This chapter show how we modified μDyner to support the extension described in the contribution chapter. However, this requires a more detailed understanding of μDyner than given in the overview before. The rest of this chapter describes the different components of μDyner. For each component, we present the modification we introduced.

5.1 Inside μDyner

The base program implements the application. The μDyner kernel library is linked at load-time to the base program through the `LD_PRELOAD` shell variable. When a program is executed, the dynamic linker `ld.so` is used to retrieve and load the shared libraries. At this point, along with the required libraries, `ld.so` will load as well the shared libraries listed by the `LD_PRELOAD` variable. This trick avoids any intrusive modifications of the base program compilation scheme. This kernel library is responsible of three tasks:

1. Load the aspect shared library to be woven in the address space memory of the base program;
2. Translate the symbolic designation of the rewriting sites provided by the aspects into addresses. The kernel library relies on meta data to perform this translation. These meta data are packaged as shared libraries generated on a per need basis by the kernel library.
3. Loading the hooking libraries in order to rewrite the pointcuts required by the aspects. A hooking library provides the ability to rewrite a specific type of site. All in all, an aspect provides the "what" to rewrite in the base program. The meta data libraries provide

the "where" to rewrite in the object code of the base program. The difference between "what" and "where" lies in the fact that an aspect can only designate the rewriting sites symbolically while an effective address is required to rewrite. The kernel library does not perform the rewriting by itself. This "how to rewrite" is provided by the hooking kernel.

Because aspects are packaged as dynamically loaded libraries, the kernel library is independent from the aspects to be woven. For similar reasons, the kernel library is independent from the structure of the base program thanks to meta data libraries. Since the actual rewriting is delegated to hooking libraries, the kernel library is as well independent from the type of pointcut and of the type of rewriting sites. This independence has been a previous asset to extend μDyner to C++.

In the next sections we will discuss the details on these libraries. We will focus on their roles and interactions between them and kernel, and also their generations. The description of the different interactions between the different components is the key to understand the weaving process used by μDyner. Based on those information, we show how we extend μDyner for C++.

5.2 Aspect Compliation

An aspect shared library contains the advice code along with and the code checking the pointcut. The shared library has been generated by the aspect compiler.

5.2.1 Interaction between the kernel and the aspect

When the application is deployed, the μDyner kernel forks a thread that waits on a socket for weaving and deweaving requests from third part processes. Upon reception of a weaving request, the μDyner kernel loads the requested aspect into the address space of the base process as a shared library (using `dlopen`) and then instruments the join points described by the innermost pointcut of the aspect. This instrumentation essentially consists in the introduction of hooks in the base program.

5.2.2 Aspect compilation

The aspect compiler generates an aspect shared library from an aspect program. It is a two-stepped process. Firstly, the aspect preprocessor translates the aspect program into C code. Next, it uses `gcc` to compile that C code to generate the aspect shared library. In this section, we will discuss them in detail and then propose our extension for C++.

The Aspect preprocessor

μDyner aspect complier is structured around aspect preprocessors. Each aspect preprocessor is responsible of translating a given type of pointcut into C code. This top level aspect compiler coordinates different preprocessors before using gcc to generate a compiled aspect. The goal of the different preprocessors is to translate the aspect code into C. Advanced users can extend the aspect syntax by providing new preprocessors. The current implementation ties a type of rewriting site to a preprocessor. It offers three kinds of aspect preprocessors - global-variable-read, global-variable-write and function-invocation.

5.2.3 Extension - A new aspect compiler

The preprocessors of the original μDyner are used without modifications in our extension. They allow to compile aspects that should be triggered on early bound function or global variables. For the other functions, we introduce a new preprocessor that generate the appropriate C++ code. We also modified the original aspect compiler provided by μDyner so that it uses a C++ compiler to compile the C++ code generated by the post-processor.

5.3 Meta-Data Library

Due to its central role, the kernel knows the pointcut sequence for any aspect and knows "what" to do at the rewriting site. The next thing is to know "where" to rewrite. In μDyner aspect system, the meta-data libraries offer this kind of information.

5.3.1 Interactions between the kernel and meta-data libraries

After the kernel receives a weaving request, it will use the meta data libraries to locate the rewriting site. For a given type of rewriting site, a meta data library contains a representation of all the rewriting sites available in the base program. μDyner uses as a rewriting site's the innermost element appearing in the pointcut sequence.

If the kernel failed to load a meta data libraries, it will uses the appropriate post processor to generate the missing meta data. Thus, meta data are generated on the fly on a per need basis.

5.3.2 Meta-data library Deployment

In order to collect the information for the meta-data, the different types of the join points associate with different postprocessors for rewriting site. That is, according to the type of the join point, the kernel will call different postprocessor for the appointed type, which are still the three types in the μDyner - `global-variable-read`, `global-variable-write` and `Function-invocation`.

5.3.3 Extension - A New Postprocessor

In the original μDyner, there are three postprocessors, one for each type of pointcut. For the newest postprocessor type - virtual functions, μDyner does not support to generate the meta-data library from virtual table. As we discussed in the previous parts, the virtual functions are all relocated in the virtual table. That is the new postprocessors should successfully select the necessary information from virtual table and mangled signature for the meta-data libraries.

Firstly, we release a basic program to achieve this purpose. The following method has this functionality.

But when we want to collect the information from vtable, we found there are some extra pointers are involved to support the runtime type. That is, if we have the call

```
ptr->z();
```

there needs to be some information associated with `ptr` available at runtime such that the appropriate instance of `z()` can be identified, found, and invoked.

When we try to print to the address of the vtable through the base program. They will have the different result. The Figure 5.1 describes these differences. It shows there are two extra pointers at the beginning of the `class1` and `class2` vtable [2]. The first two pointer are pointing to the offset of the class and the RTTI type of its super class. This is only the single inheritance case.

When it comes to the multiple inheritance and virtual inheritance, the problem will be more difficult. If we check the beginning address of each vtable and each VTABLE[0]. We will find the difference is not 8 bytes anymore.

[2] gives the reasons of this difference. We examine them one by one. Firstly, the top class `A`, its vtable is the same as the one of `class1`.

```
A::offset_to_top (0)
A::rtti
```

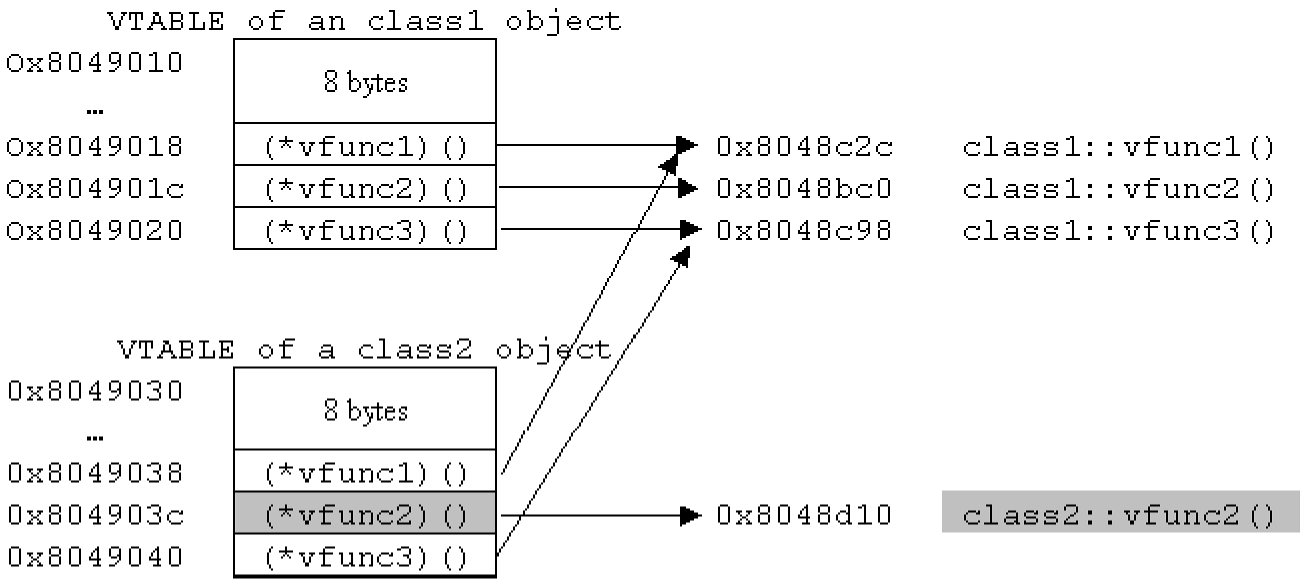

Figure 5.1: Vtable layout of class1 and class2

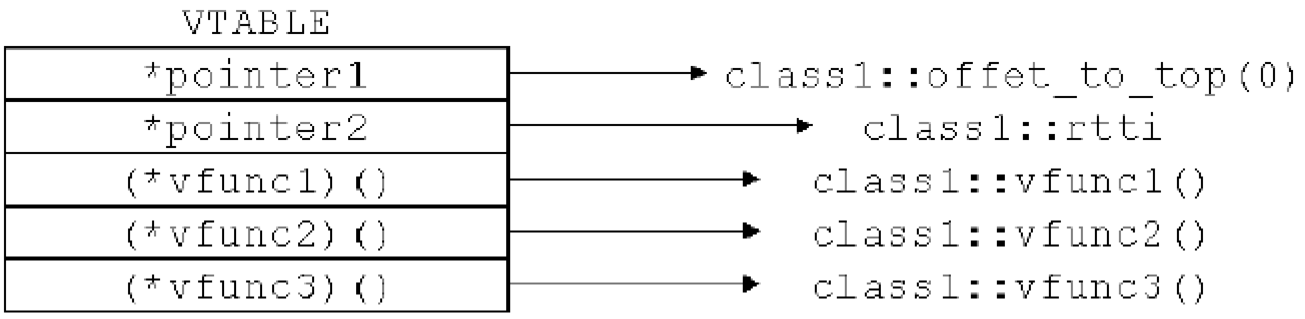

Figure 5.2: The pointers inside of vtable of virtual function

Class Name	Beginning Address	VTABLE[0] Address	Size	Difference	Number of Pointers
A	0x804a4d8	0x804a4e0	14H (20D)	8H (8D)	5
B	0x804a4a0	0x804a4c8	34H(52D)	28H (40D)	13
C	0x804a460	0x804a488	34H(52D)	28H (40D)	13
D	0x804a400	0x804a43c	48H(72D)	3cH (60D)	18

Table 5.1: The difference between the beginning address and VTABLE[0]

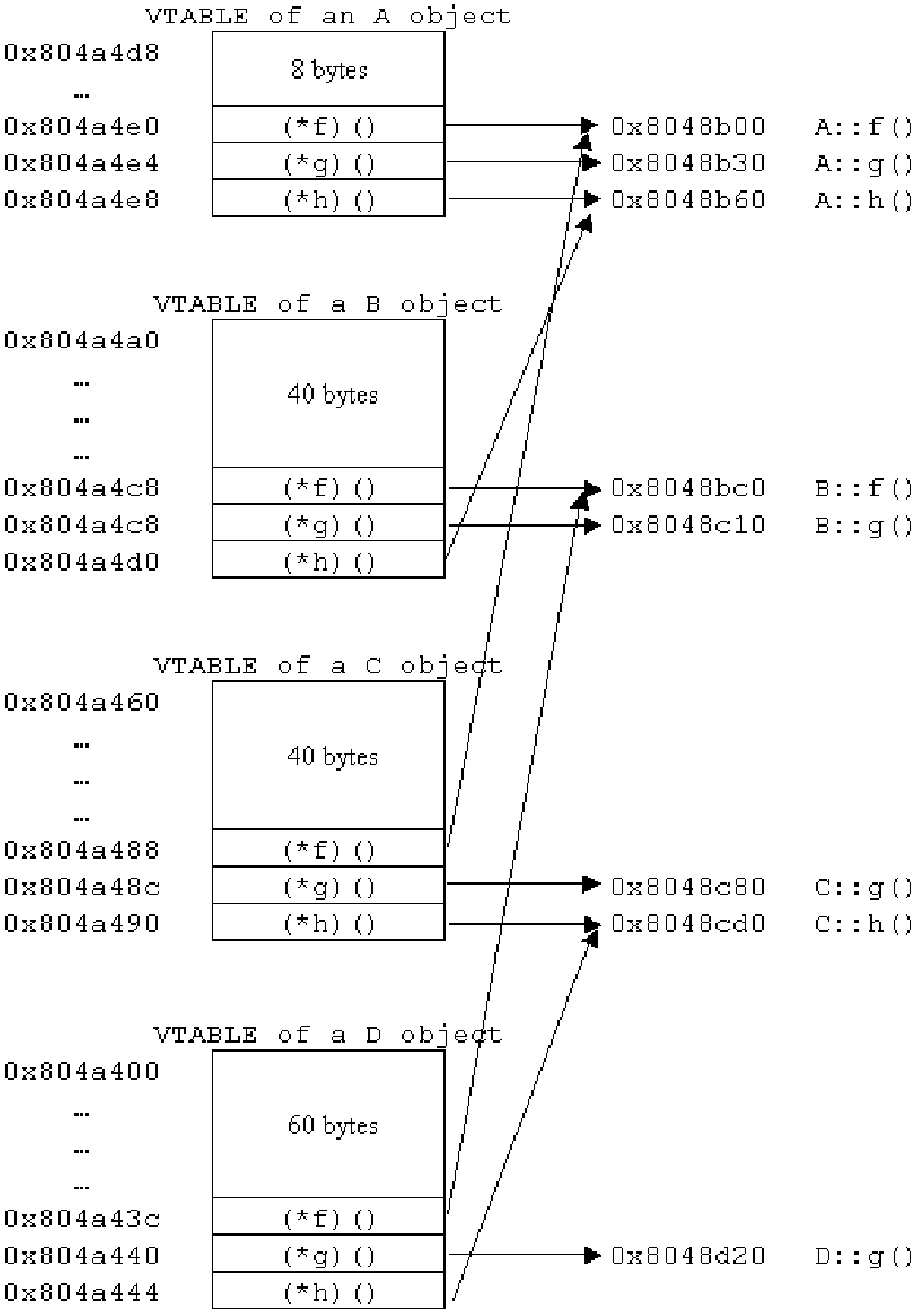

Figure 5.3: Virtual function pointers of vtable

```
A::f() []
A::g() []
A::h() []
```

In B, it overrides the functions `f()` and `h()`. As A is the virtual base class of B, there are more extra pointer are added into the vtable of B. The information is followed.

```
B::offset_to_A (16)
B::offset_to_top (0)
B::rtti
B::f() []
B::g() []
A::offset_for_h (-16)
::offset_for_g (0)
A::offset_for_f (-16)
A::offset_to_top (-16)
A::rtti
B::f() [[-24] offset_for_f]
A::g() []
B::h() [[-24] offset_for_h]
```

In C, it overrides the functions `g()` and `h()`. The same as B, there are more extra pointer are added into the vtable of B. The information is followed.

```
C::offset_to_A (16)
C::offset_to_top (0)
C::rtti
C::g() []
C::h() []
A::offset_for_h (-16)
A::offset_for_g (-16)
A::offset_for_f (0)
A::offset_to_top (-16)
A::rtti A::f() []
C::g() [[-32] offset_for_g]
C::h() [[-40] offset_for_h]
```

In D, it overrides the functions `h()`. The same as B, there are more extra pointer are added into the vtable of B. The information is followed.

```
D::offset_to_A (32)
D::offset_to_top (0)
D::rtti
B::f() []
D::h() []
C::offset_to_A (16)
C::offset_to_top (16)
C::rtti C::g() []
D::h() [-16]
A::offset_for_h (-32)
A::offset_for_g (-16)
A::offset_for_f (-32)
A::offset_to_top (-32)
A::rtti B::f() [[-24] offset_for_f]
C::g() [[-32] offset_for_g]
D::h() [[-40] offset_for_h]
```

The difference between the virtual table belonging to the virtual functions and its belonging to the virtual base class is the latter have more extra pointers in the derived D in our case.

This is the details of the pointers in virtual table. When we know the details of the layout of those virtual tables. It is very essential to our implementation part.

5.4 Hooking Library

Weaving with μDyner requires the identification of the join points in the executable code and modifying the join points to jump to the code implementing the advice. The hooking library provides these capabilities. It tells the kernel "how" to rewrite.

5.4.1 Interaction between the kernel and the hooking library

The kernel delegates the rewriting task to hooking libraries. Hooking library will rewrite according to the types of rewrite site. Thus, there are three different kinds of the hooking libraries. The hook manages the invocation of the advice Hooks check whether the sequence of pending return points on the execution stack corresponds to the sequence of pointcuts described by the aspect, and if so, invokes the advice. If the pointcuts are not satisfied, the hook performs the action of join point and returns control to the application. Corresponding to the different types of the rewriting sites, we use the different hooking libraries to generate the hooks. The interface of the concrete hooking library is as Figure 5.4

```
struct ConcreteSite{
    ConcreteSite* (*make) (char *symbolName, int
    address, int length, char *SourceFile, int sourceLine, char
    *inFunction, int inFunctionAddress,...)
    void (*serializedAsCCode)(File *file, ConcreteSite *data)
    void (*writeIncludesSupportingSerialization)(File *file,
    ConcreteSite *data)
    void (*destory)(ConcreteSite *site)
    void (*rewriteAsJump)(ConcreteSite *site, void *addressToJumpTo,
    Guard *guard)
    void (*restoreFunctionCode)(ConcreteSite *site)
    boolean (*equals)( ConcreteSite *aSite, ConcreteSite *anotherSite)
    boolean (*hasBeenRewritten)(ConcreteSite *site)
}
```

Figure 5.4: The internal representation of concrete hooking library

5.4.2 Extension - A New Hooking Library

The existing μDyner let the global variables and the functions can be hooked. In μDyner for C++, we focus on functions, instance functions, virtual functions. There is not virtual function pointcut in the original μDyner. For a virtual function, the hooking library will directly change the pointer in the vtable. The following program proves this idea is feasible.

```
    void changeVT (  class1  *pa ) {
        // p sees pa as an array of dwords
        unsigned * p = (unsigned *)(pa);
        // vt sees vtable as an array of pointers
        int ** vt = (int **)(p[0]);
        printf( "vtable[0] address = %p\n",vt);
        printf("%p %p\n",vt[2],&vt[2]);
        vt[2]=(int*)(advice);
    }
```

In this method, the input is the pointers of `class1`, which is an example in the previous chapter. The first pointer is `vptr` of `class1`. Through this `vptr`, we can access to the vtable. With the strict order rules in vtable, the method `class1::vfunc3()` is pointed by the pointer at VTABLE[2]. If the input is the pointers of `class2`, a derived class of `class1`, the pointer at VTABLE[2] is also successfully changed to pointing `advice()`.

After knowing we can replace the pointers in the vtable, our hooking library will be built on this idea. When it comes to the other two pointcuts type - function and instance function, the μDyner uses the original technique. More, for the instance function joinpoint, the hooking libraries should solve the scope operator.

5.5 Summary

This Chapter presented how we extended μDyner to support C++. Concretely, we modified the μDyner kernel so that it does not assume that symbols in the binary level file would the number of the function appearing in the pointcut aspect source code. We changed the manner μDyner retrieves the symbol name to take into account the C++ name mangling scheme. We changes the aspect compiler so that advice can be programmed in C++. We introduced a new prepocessor handling pointcut ending with virtual function call. We had a new post-processor to the system collecting vtable information. Finally, we implemented a new hooking library allowing aspect to be triggered on virtual function call.

Chapter 6

Conclusion

The Web latency has become an important issue. While prefetching appears as a low cost solution, the efficiency of prefetching largely depends of the user access patterns and of the contents provided by servers. For maximum efficiency prefetching policies should be deployed on demand in Web caches.

Web caches like Squid are large C programs. Squid architecture relies on a set of statically linked modules. In this particular architecture prefetching policies are crosscutting concerns. Therefore, it has been proposed to use dynamically woven aspects to deploy prefetching policies in Squid. A tool has been designed in this goal: μDyner. It provides an aspect system for C program.

But future version of Squid will be written in C++. This dissertation explored the possiblity to apply the weaving techniques devised for the C language in μDyner to the C++ language. The basis of these techniques is to rewrite on the the fly the object code executed by the microprocessor.

Our contribution is twofold. First of all we highlight key elements that the pointcut language should cover. Second, we conclude that the techniques devised for C can successfully be applied to C++. The latter conclusion was difficult to establish due to the number of technical details that had to be considered.

As a side effect and despite the lack of time to realize a detailed performance study, it should be noted that our extension of μDyner for C++ should yield good performance. In particular, due to late binding, weaving an aspect executed on a virtual function call, reduces to an exchange of pointers.

Much work is still left. First of all, it would interesting to implement prefetching policies in Squid using our extension as soon as a C++ version of Squid will be available. This could lead to reshape the aspect language.

Another interesting point would be to package the modifications we made to the original μDyner as aspects. These aspects could be woven on demand, when μDyner detects - using the debugging - information that the base program has been written in C++. This approach would result in a first step towards a language independent weaving infrastructure. In the second step, we could like to study the possibility to offer an aspect language decoupled from the programming language used for the base program. We do believe that this could support "aspect-off-the-shelf".

Bibliography

[1] http://www.alphaworks.ibm.com/tech/hyperj.

[2] http://www.codesourcery.com/cxx-abi/css-vtable-ex.htr.

[3] Noury M.N. Bouraqadi-Saadani and Thomas Ledoux. How to weave? *EMN Technical report*, 2001.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[5] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[6] Constantinos A. Constantinides and Tzilla Elrad. On the requirements for concurrent software architectures to support advanced separation of concerns. *The Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, 2000.

[7] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Measurement and Modeling of Computer Systems*, pages 178–187, 1999.

[8] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. *Lecture Notes in Computer Science*, 2170:556+, 2001.

[9] Kiczales G. and al. Aspect-oriented programming. *In proceeding of ECOOP'97,LNCS 1241, Springer-Verlag*, 1997.

[10] Andreas Gal, Wolfgang Schroder-Preikshat, and Olaf Spinczyk. Aspectc++: Language proposal and prototype implementation. 2001.

[11] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[12] Zhimei Jiang and Leonard Kleinrock. Prefetching links on the WWW. In *ICC (1)*, pages 483–489, 1997.

[13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *In proceeding of ECOOP'01*, 2001.

[14] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[15] G. Abdulla S. Williams M. Abrams, C.R. Standridge and E.A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the Fourth International WWW Conference*, 1995.

[16] Liangjing Zhao Marc Ségura-Devillechaise, Jean-Marc Menaud. Patchwork: an aspect-oriented approach to unanticipated adaptation of web caches, 2003.

[17] Jean-Marc Menaud. *Cooperative Cache Systems For Large Scale Distributed Information System*. PhD thesis, IRISA/INRIA Rennes, France, 2000.

[18] Themistoklis Palpanas and Alberto Mendelzon. Web prefetching using partial match prediction. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[19] D. L. Parnas. A course on software engineering techniques. *In Papers of the second ACM SIGCSE symposium on Education in computer science*, 16:154–159, 1972.

[20] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. *In proceeding of 1st International Conference on Aspect-Oriented Software Development (AOSD), Entschede, Netherlands*, April 2002.

[21] Wolfgang Schult and Andreas Polze. Speed vs. memory usage - an approach to deal with contrary aspects, journal=The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at the International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, year=March 17 - 21, 2003,.

[22] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution, 2003.

[23] Lionel Seinturier, Renaud Pawlak, Laurence Duchien, and Gërard Florin. Jac milestone 2001. *A joint research project between the CEDRIC-CNAM and LIP6 computer science laboratories*, September 2001.

[24] D. Wessels and contributors. Squid web proxy cache. http://www.squid-cache.org/.

[25] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna R. Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

[26] Wei Shu Yingyin Jiang, Min-You Wu. Web prefetching: Costs, benefits and performance, August 14-16, 2002.

[27] Charles Zhang and Hans-Arno Jacobsen. Towards building a dynamic weaving aspect language for c. *In foundation of Aspect-Oriented Languages Workshop in conjunction with 2nd AOSD Conference, Boston, MA*, 2003.