Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution

Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller
Obasco group, Ecole des Mines de Nantes/INRIA
4, rue Alfred Kastler, La Chantrerie, Nantes, France
msegura,jmenaud,gmuller@emn.fr

Julia L. Lawall
DIKU University of Copenhagen
2100 Copenhagen, Denmark
julia@diku.dk

ABSTRACT

Given the high proportion of HTTP traffic in the Internet, Web caches are crucial to reduce user access time, network latency, and bandwidth consumption. Prefetching in a Web cache can further increase these benefits. Nevertheless, to achieve the best performance, the prefetching policy used must match user and Web server characteristics. This implies that new prefetching policies must be loaded dynamically as needs change.

Most 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 is a natural technique to address this issue. Nevertheless, existing approaches do not provide dynamic weaving of aspects targeted toward C applications. In this paper, we present μDyner, which addresses these needs. μDyner also provides lower overhead for aspect invocation than other approaches, thus meeting the need for good performance in Web caches.

Keywords

Adaptable software, aspect-oriented programming, code instrumentation, pointcut language, Web caches

1. INTRODUCTION

Because HTTP amounts, at the very least, to 80% of the Internet traffic [17], caching HTTP documents is an appealing approach to decrease the Internet latency and network bandwidth consumption [35]. Two factors, however, decrease Web cache effectiveness: (i) between 35% to 50% of Web documents are uncacheable because their content is specific the initial request [35], (ii) once cached, many Web documents are never requested again [25].

A strategy to overcome these limitations is to prefetch Web documents so that they are already in the cache and thus close to the client when first requested [19]. Nevertheless, a simple prefetching strategy such as prefetching all of the documents reachable from a document requested by the user only reduces the user access time for the few such pages that are actually referenced, at the expense of increasing the overall bandwidth consumption and the workloads of Web servers. Instead, strategies that are tailored to user preferences and Web server characteristics are needed to ensure that most of the prefetched documents are eventually accessed. For example, Issarny *et al.* have shown that in the context of an electronic newspaper, a prefetching policy that is based on user profiles and specialized to the targeted Web server can achieve a prefetch prediction accuracy of up to 92% [18]. To support the use of such policies, a Web cache should be extensible and support dynamic loading and unloading of new policies.

Many Web caches, such as Squid [12], 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. Unfortunately, prefetching policies often crosscut the functionalities of several modules. Thus, a prefetching module would have to redundantly implement many basic cache operations.

In fact, the crosscutting nature of prefetching policies suggests that such policies should be implemented using aspect-oriented programming (AOP). Nevertheless, Web caches possess specific characteristics that motivate the need for a specific AOP infrastructure, providing the following features:

- 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 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.

This paper

This paper describes the μDyner AOP infrastructure for writing and dynamically deploying aspects in running C programs without inducing service unavailability. The design of μDyner targets the adaptation needs of Web caches. More precisely, our contributions are as follows:

- We demonstrate that prefetching policies can naturally be implemented using aspects.
- We provide an approach to insert aspects at run time into an application written in C. Unlike approaches based on Java that rely on JIT compilation for good performance, our approach manipulates only native code at run time, and thus directly produces executable code.
- The cost of calling a null function dynamically augmented by a null aspect is 9.5 times higher than that of calling a null function with no woven aspect; similar experiments with Java-based solutions show an overhead of 20-70 times.
- Weaving a new aspect requires only a few hundred microseconds, and much of this process is transparent to the application. Thus, our approach induces little freeze time of the application.

The rest of this paper is structured as follows. Section 2 presents an overview of Web caches and issues in implementing prefetching. Section 3 describes the μDyner framework. Section 4 evaluates the cost of inserting a μDyner aspect. Section 5 presents related work. Section 6 concludes and describes future work.

2. WEB CACHES AND PREFETCHING

This section first describes the software architecture of a typical Web cache, such as Squid [12]. Then, we present issues in integrating a prefetching policy into an existing cache. Finally, we show that AOP conveniently addresses these issues.

2.1 Web cache architecture

A Web cache has three goals:

- for the end users: to decrease the average access time,
- for the organization managing the Web cache: to decrease the bandwidth consumption,
- for the ISP: to decrease the workload on each server encountered.

The basic behavior of a Web cache is as follows. A Web cache sits between users and Web servers and intercepts user requests. On the receipt of a request, the Web cache checks whether the requested document is already in its local storage. If so, the cache sends the document to the user directly. Otherwise, the cache forwards the request to the Web server, downloads the document to its local storage, and returns it to the user. When the local storage of the cache is full, the cache's replacement policy is activated to remove potentially useless documents.

To improve effectiveness, several Web caches may be associated by means of a cooperation protocol such as ICP [34]. When such a *cooperative cache* does not already possess the requested document, it first attempts to find it on one of its associated neighbors before forwarding the request to the server. This architecture relies on the assumption that communication between caches is much faster than communication with the server.

Web caches are often implemented as a collection of modules, each implementing a single functionality [2, 12, 36]. As illustrated in Figure 1, these modules implement three basic functionalities: 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 for the document in the local storage (3 and 4), and possibly forwards the request to the Web server (6) or to its neighbors (7). If a new document is obtained, it is saved to the local storage (4), which may require an activation of the replacement policy (10). Finally, the document is sent to the user (8). To communicate with a neighbor, the cache accepts and parses requests from the neighbor (5), checks whether the requested document is locally available (3 and 4), and answers the neighbor with either an error message or the requested document (9).

2.2 Prefetching

The implementation of a prefetching policy must address several issues that impact diverse parts of the Web cache:

- The kinds of incoming requests to which prefetching is applied: To avoid overloading the network and overflowing the storage space of the Web cache, prefetching should only be applied to user requests, not to requests from neighbors. Thus, the prefetching policy must be aware whether the request was received by the user-request module or the ICP-request module of the cache.
- The choice of documents to prefetch in response to a user request: Possible approaches include fixed strategies such as prefetching a few links near the top of the document [6] and statistics-based approaches that require the Web cache to maintain and analyze a history of its transactions [9].
- The hosts that are queried to find prefetched documents: Querying only the Web server limits the increase in network bandwidth due to prefetching. Querying the neighbors can produce a result more quickly. Whichever strategy is taken, the network module of the Web cache must be aware that prefetch requests should be treated differently than ordinary user requests.
- The lifetime of prefetched documents within the storage space of the Web cache: Prefetched documents should remain in the cache long enough to have a reasonable chance of being accessed, and thus should not be the highest priority candidates of the Web cache's

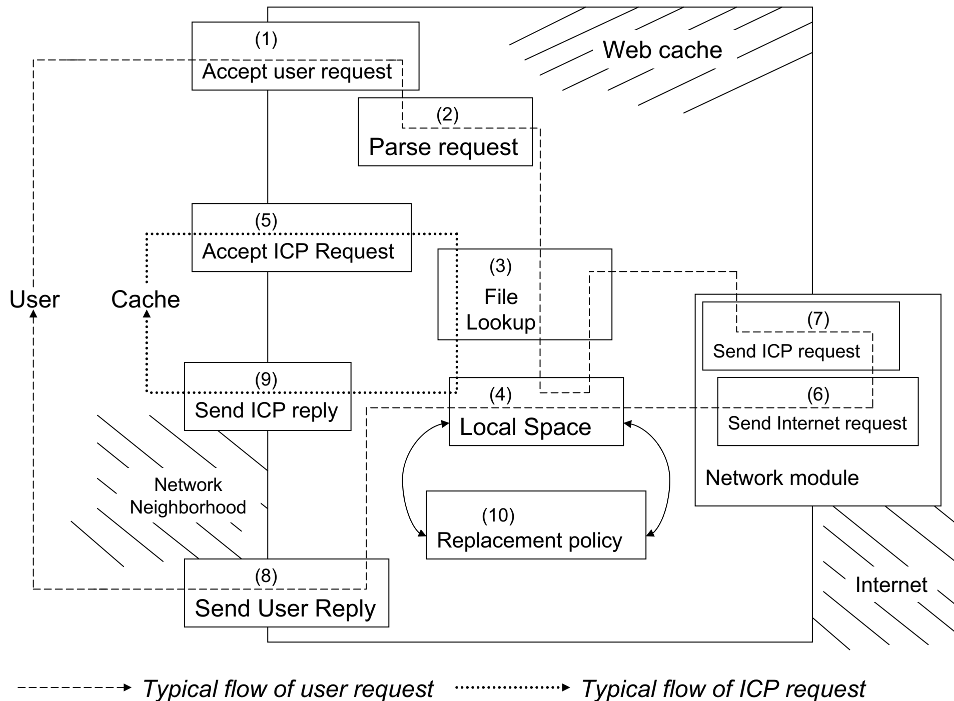

Figure 1: Structure of a modular Web cache

replacement policy. On the other hand, when a document is removed from the cache, the replacement policy should remove documents prefetched for that document as well.

One approach to add prefetching to an existing Web cache is to follow the model of a Web cache itself and use an interposition approach. That is, we add a new server in front of the Web cache that implements the prefetching policy and delegates all other caching tasks to the Web cache. Nevertheless, this approach is inadequate in the case of prefetching because of the significant need for communication between the prefetching policy and the Web cache. For example, the prefetching policy must instruct the Web cache whether to request prefetched documents from its neighbors, and the Web cache must inform the prefetching policy when documents are removed from the cache by the replacement policy. Thus, the approach of implementing the prefetching policy in a separate server does not seem promising.

The Apache [2] and MOWS [36] Web caches provide extensibility using dynamic loading and unloading of modules.[1] At specific points in the treatment of a request, these systems check whether a module is able to take over the computation. If there is such a module, it is then responsible for all subsequent treatment of the request. This approach is not well suited to the implementation of prefetching, because it implies that the prefetching policy must reimplement a significant subset of the Web cache functionality. Indeed, it is rather difficult to define a generic interface that modules should export in order to support prefetching. In fact, the kinds of interactions that are needed are highly dependent on the behavior of prefetching policy.

The problem of the degree of module granularity and the difficulty of defining an appropriate interface suggest that prefetching could be implemented in simpler manner using aspect-oriented programming.

2.3 Prefetching as a collection of aspects

We now illustrate how a prefetching policy can be implemented using aspects. As an example, we use the *Interactive Prefetching* policy of Chinen and Yamaguchi [6] that prefetches a few documents referenced from the top of the requested document as well as all of the images required for these documents. For concreteness, we consider the implementation of this policy in a cooperative cache, for which we specify that prefetched documents should only be obtained from the Web server. We alse assume a LRU replacement policy. In describing the aspects needed to implement this policy, we use the following AOP notions [21]:

- Join points: A join point is a point in the code that can be modified by an aspect.
- Pointcuts: A pointcut is a description of the execution contexts in which an aspect should be activated.
- Advice: Advice is the code implementing the functionality provided by an aspect.

The Interactive Prefetching policy selects the documents to prefetch based on information contained within the document itself. This implies that prefetching for new documents can only be initiated after the network module, at

[1] Squid does not support dynamic loading of modules.

which point the contents of the document have been received. Aspects implementing prefetching, however, should only be activated if the request is a user request, rather than a neighbor or prefetching request. Because this information is no longer part of the control stack at the point when the document is received from the network, pointcuts are not sufficient to distinguish between these cases. Instead, this information can be recorded in a new hash table maintained by the prefetching policy that maps each request identifier to the source of the request. The user request module and the ICP request module must both be adapted with advice that updates this hash table.

Prefetching should also be initiated when there is an incoming request for a document that is found in the cache, but for which prefetching has not previously been applied (as would be the case for a prefetched document, for example). Again, prefetching should only be applied when the request comes from the user. In this case, pointcuts can be used so that the advice is only invoked when the file lookup module is called from the user request module.

A LRU replacement policy relies on document access time 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 requested document, and not on the access time of the prefetched document itself. Thus, the local space module must be adapted with an aspect that identifies new prefetched documents and sets their access times accordingly. The file lookup module must similarly be adapted with an aspect that updates the access times 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.

Many other prefetching strategies have been developed.[4, 9, 8, 14, 26, 33] These strategies vary in how they crosscut a Web cache implementation. For example, prefetching decisions can be made based on statistical observations about the history of incoming requests [9]. The implementation of such a strategy more deeply crosscuts the module that accepts user requests and the modules that manage the storage space than does the implementation of the Interactive Prefetching policy. An alternative implementation of this strategy is for Web servers to maintain these statistics. In this case, a piggybacking protocol must be established with the server, so that it can transmit prefetching hints to the Web cache. The implementation of this strategy thus crosscuts the network module. Rather than prefetching documents, the Web cache can simply pre-establish a connection with the Web server [8], thus reducing the latency perceived by the user. Such a strategy crosscuts the Network module, but in a different way than the use of a piggybacking protocol.

3. OVERVIEW OF μDYNER

The μDyner aspect system provides the ability to dynamically weave and deweave aspects in executing C applications. Three types of users interact with μDyner: the maintainer of the base code, the aspect developer, and the cache administrator. The maintainer of the base code is responsible for annotating the cache implementation to indicate the points at which adaptation is allowed. The aspect developer is responsible for identifying appropriate prefetching algorithms, writing the corresponding aspect code, and compiling these aspects using the μDyner dedicated compiler. Ideally, the Web cache should monitor incoming requests and select appropriate prefetching policies as conditions change. In a first step, to validate our approach, aspects are installed manually by a cache administrator.

3.1 Modifying a cache to support aspects

μDyner allows weaving of aspects at run time. To avoid costly runtime decompilation and reorganization of the code of the running application, the μDyner approach relies on a source instrumentation of the program to prepare for later adaptation. Specifically, μDyner provides a source-level annotation `hookable` with which the program maintainer annotates the points in the program at which adaptation is allowed. Only global variables and functions can be declared to be `hookable`.

Besides the practical benefit of reducing the cost of adaptation, the use of the `hookable` annotation also gives the base program maintainer, who knows the structure of the code best, some degree of control over the subsequent adaptation. In particular, the maintainer is aware of critical segments of the code in which adaptation should not be allowed, and can thus ensure that no hookable constructs are mentioned in these segments.

The base program maintainer must also link the application with the μDyner instrumentation kernel. When the application is deployed, this kernel forks a thread that waits on a socket for weaving and deweaving requests from the cache administrator.

3.2 Aspect development

The aspect developer defines the pointcuts and the advice needed to implement the prefetching policy. Based on this information, the μDyner compiler produces executable code that drives the instrumentation process.

The syntax of the aspect language is defined in Figure 2. The definition of an aspect consists of its name and a nested sequence of pointcuts that describe the affected join point. Each pointcut but the last must describe a function invocation (`function-invocation`), and the sequence represents a sequence of direct nested calls. The innermost pointcut can either be another function invocation or a global variable access (`global-variable-access`). The latter case describes a variable access that occurs in the body of the function mentioned in the innermost `function-invocation`, or anywhere in the program if no `function-invocation` is mentioned. The sequence of pointcuts ends with an advice, which is implemented as an ordinary C statement.

The advice associated with an aspect is executed when the current execution context matches that described by the sequence of pointcuts. Execution of the advice replaces execution of the join point represented by the innermost point-

```
<aspect> ::= <name> ":[" <filters-advice> "]"

<name> ::= <identifier>

<filters-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-read> ::= <type> <identifier>

<global-variable-write> ::= <type> <identifier> "=" <identifier>

<advice> ::= <C-compound-instruction>
```

Figure 2: The aspect language.

```
prevent_propagation :[
    int handle_request(char * req) :[
        int relay_request(struct req_data * request) :[
            { #include "prefetching.h"
              if (is_prefetching_request(request)) {
                 return NO_NEIGHBOR_HAS_FILE;
              }
              return continue_relay_request(request);
            }
        ]
    ]
]
```

Figure 3: An extract of a prefetching policy.

cut. Advice runs in the same address space as the base program, and thus can refer to the base program's global variables. If the advice replaces a function call, the advice can also refer to the arguments of this call, via the parameter names declared in the innermost pointcut. The advice can furthermore call the replaced function using the implicitly generated function `continue_<function_named-_in_the_pointcut>`. A pointcut representing an assignment (`global-variable-write`) includes both the name of the affected variable and a new variable representing the value of the right-hand-side expression. This new variable can also be referred to by the advice. In all cases, the advice can maintain local state across invocations using C `static` local variables. When the execution of the advice completes, its return value is returned to the base program as the result of the join point execution.

Figure 3 presents an aspect from the implementation of a prefetching policy that prevents the propagation of a request to cache neighbors. This aspect assumes that the Web cache uses a function `handle_request` to handle requests, and that propagation of a request to the neighbors is implemented by the function `relay_request`. The aspect `prevent-propagation` replaces a call to `relay_request` from `handle_request` by advice that checks whether the request, obtained as the argument to `relay_request`, is a prefetching request. If so, the advice returns `NO_NEIGHBOR_HAS_FILE`, indicating to the Web cache that it must request the document from the Web server. If the request is not a prefetching request, the advice invokes the original definition of `relay_request`, using the function `continue_relay_request`. To allow the weaving of this aspect, the source code need only declare the function `relay_request` to be `hookable`.

3.3 Aspect deployment

In the current stage of our experiments, aspects are deployed by the administrator of the Web cache. μDyner provides two commands `weave` and `deweave` to deploy and undeploy aspects. The command `weave` has two forms. The form:

```
weave <pid> <aspect-weaver> ...
```

weaves the compiled aspects `aspect-weaver ...` into the host process identified by `pid`. The form:

```
weave <pid>
```

lists the names of all of the aspects currently woven into the process `pid`. The command `deweave` has the form:

```
deweave <pid> <aspect-name> ...
```

If any of the aspects named `aspect-name ...` are currently woven into the process identified by `pid`, the aspects are dewoven. Otherwise, `deweave` returns immediately.

3.4 Implementation of μDyner

Figure 4 illustrates the effect of weaving the aspect of Figure 3 into a Web cache process. We now describe how dynamic weaving and deweaving are implemented by μDyner.

Compile-time processing

Much of the process of dynamic weaving is prepared at compile time, including processing of the source program and processing of the aspects. This compile-time processing allows weaving at run time to be very efficient.

Weaving is performed directly on the executable image of the host program, which is currently loaded into memory. This process requires the ability to identify join points in the executable code and the ability to modify the join points to jump to the code implementing the advice. The `hookable` annotations cause the code to be compiled in a manner that ensures these capabilities, as described below.

A function-call join point is implemented by modifying the function definition rather than the call sites. Detecting the position at which advice should be added is thus straightforward: a function is compiled as a sequence of instructions at an offset from the beginning of the executable image that is determined at link time. The `hookable` attribute on the function is used to ensure that there is sufficient space at the beginning of the function to insert a jump to the advice before the actual compiled function body. Specifically, `hookable` expands to a branch instruction to the actual start of the function followed by `NOP` instructions if needed to fill the remaining reserved space. When advice is added, the reserved space is overwritten by a jump to the advice, as shown in Figure 4, and the address of the actual function body is stored in the definition of the implictly generated `continue_<function_name>` function. The `hookable` attribute also prevents the function from being inlined.

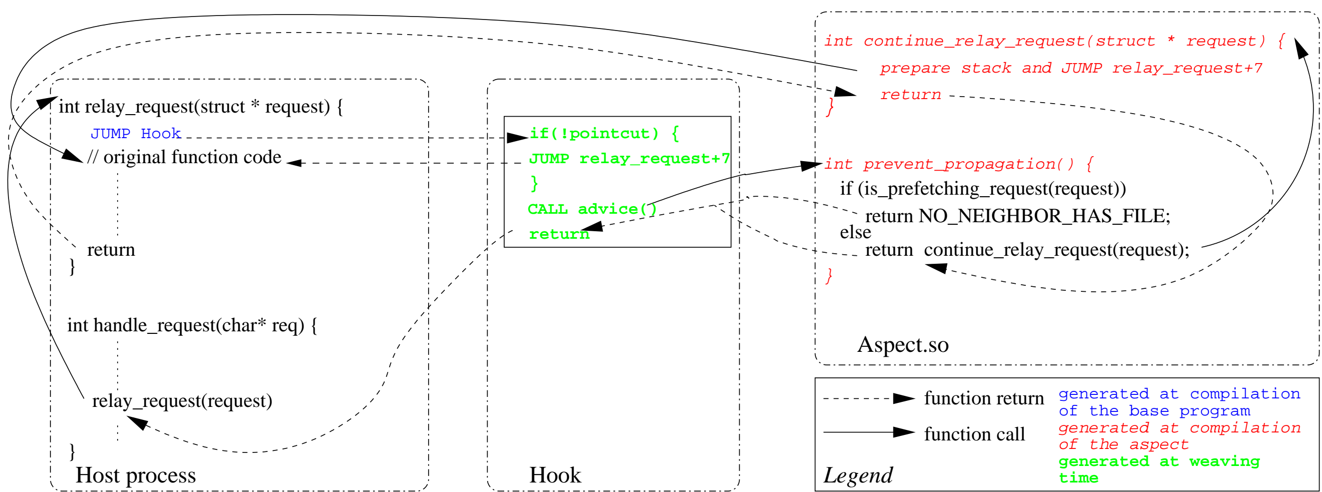

Figure 4: Execution of an aspect.

A variable-access join point is implemented by modifying each access to the variable. In general, a variable can be implemented as a memory location or as a register, or indeed as both. Because it is non-trivial to track accesses through registers, μDyner requires that a variable-access join point be implemented as an access to an explicit memory location. The `hookable` annotation on a variable declaration thus macro-expands into a `volatile` declaration, which forces the compiler to implement every access to the variable as an explicit access to the associated memory location.[2]

For each aspect, the μDyner compiler generates code that checks whether the current execution stack matches the specification of the pointcuts. The μDyner compiler also invokes the C compiler to generate code for the advice.

Weaving

The weaving function of μDyner sends a message to the socket associated with the μDyner instrumentation kernel in the host process. On receipt of this message, the μDyner kernel loads the requested aspect into the address space of the host process as a shared library (using `dlopen`) and then instruments the join points described by the innermost pointcut of the aspect. This instrumentation consists of creating *hooks* that manage the invocation of the advice and updating each join point with a call to the associated hook. A hook checks 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 the join point and returns control to the application. The exact behavior of the hook depends on whether the join point is a `function-invocation`, `global-variable-read`, or `global-variable-write`, as described below. Because the definition of a hook depends on both the base program and the aspect, hooks are created dynamically.

[2]This approach is insufficient to detect references to a variable via a pointer. The μDyner compiler gives an error if the source program ever takes the address of a variable declared as `hookable`.

As shown in Figure 4, a single hook is used for an aspect whose innermost pointcut is a `function-invocation`. This hook first saves the complete set of registers of the processor and then checks that all of the pointcuts for the aspect are satisfied. If they are, the hook calls the advice on the arguments given to the function. The result returned by the advice is itself returned by the hook. Because the hook is invoked by a jump instruction rather than by a function call, the return address on the stack at this point is the return address of the call to the join point function, so control returns to the call site rather than to the original function body, as shown in Figure 4. If some of the pointcuts of the aspect are not satisfied, the hook restores the registers and jumps back to the function body.

When the innermost pointcut of the aspect is a `global--variable-read`, μDyner creates a hook for each reference to the variable. Each such hook is instantiated with the address of the variable reference and the instruction formerly present at that address, and thus is able to continue the computation from the point of the variable reference. As for a hook for a `function-invocation` join point, the hook for a `global-variable-read` join point initially checks that all of the pointcuts for the aspect are satisfied. If so, the hook invokes the advice. Because all of the global variables of the program are visible to the advice, there is no need to pass the address of the variable associated with the aspect to the advice. When the advice completes, the effect of the original variable-reference instruction is applied to this value, and the hook jumps back to the instruction following the join point. If the pointcuts of the aspect are not all satisfied, the original variable reference instruction is executed and the hook jumps back to the instruction following the join point.

The treatment of a `global-variable-write` hook is similar to that of a `global-variable-read` hook except that in this case, the advice must be able to access the value of the right-hand side expression of the assignment, via the variable name specified in the pointcut. If the pointcuts of the

aspect are satisfied, the hook examines the instruction implementing the assignment in the original program to obtain the value of the right-hand side expression. The hook then passes this value to the advice.

Deweaving

Deweaving an aspect simply overwrites the code at each join point with its original instruction, frees the space allocated for the hook, and unloads the aspect, using `dlclose`.

4. PERFORMANCE EVALUATION

To evaluate the cost of dynamic weaving in μDyner, we have measured the overhead introduced by the `hookable` annotations, the cost of weaving a new aspect, and the cost of invoking an aspect once woven. Our experiments were conducted on an Intel Pentium 4 running at 1.6 GHz with 256 MB of RAM under Linux (kernel 2.4.17, gcc 3.0.4 with the option `-O2`).

The `hookable` annotation introduces a short branch and `NOP` instructions at the beginning of each function that is a potential join point, and causes each access to a global variable that is a potential join point to be implemented as an access to a memory location. To assess the overhead introduced by the `hookable` annotation, we compare the performance of the `quicksort` program presented in Appendix A with and without `hookable` annotations on functions and (globalized) variables in the program's critical path, when applied to an array of 10 million random integers. We found that no overhead was introduced by the `hookable` annotation as compared to using ordinary functions and global variables. This result suggests that it is feasible to place `hookable` annotations throughout the program in order to allow a wide range of possible adaptations.

Weaving an aspect into the base program requires loading the aspect and instrumenting the affected join point(s). The cost of loading the aspect is dominated by the cost of `dlopen`; loading the aspect of Figure 5 requires 175 μs. The instrumentation process replaces the join point with a jump instruction and creates the hook. The cost of writing a single jump instruction is minimal. Creating a hook for a `function-invocation` join point requires allocating space for the code (using `malloc`) and copying the code, which is pre-generated by the μDyner compiler, from the compiled aspect into this space (using `memcpy`). The cost of these operations is proportional to the number of pointcuts. For the trivial aspect of Figure 5, the total instrumentation cost is 19μs. Creating a hook for a `global-variable-access` join point additionally requires instantiating the precompiled hook with the contents and address of the instruction at the join point. The cost these extra operations is, however, minimal. Note, though, that a distinct hook is created for each access (reference or update, depending on the pointcut) to the variable, and thus the instrumentation cost must be multiplied by the number of such accesses.

Deweaving essentially inverses the process of weaving. Each of the operations involved is equally or less expensive than its weaving counterparts. For example, unloading the aspect with `dlclose` requires only 119 μs. Thus, the cost of deweaving is lower than the cost of weaving.

A single hook is installed atomically from the viewpoint of the host program, and so the application need not be stopped in this case. Nevertheless, the power of aspects comes from the ability to make changes throughout the program. When multiple hooks or multiple aspects are needed, the administrator may need to stop the host process during weaving, if there is the possibility of incorrect interactions within incompletely installed code. Nevertheless, even in this case, the process need only be stopped during the updating of the host process code with jump instructions. The more time-consuming loading of aspects and construction of the hooks can proceed in parallel with host process execution.

The cost of weaving an aspect should be compared to the retransmission timeout [27] in TCP. Typical values on a BSD implementation of TCP vary between 0.5 and 1 second [1]. It has been argued that the timeout should not be lower than 250ms [1]. This suggests that μDyner is efficient enough to enable dynamic adaptation of web caches.

```
empty :[
    int trivial() :[
            {
              return continue_trivial();
            }
    ]
]
```

Figure 5: A null aspect.

To assess the overhead of calling a function adapted by an aspect, we consider a function `trivial` that immediately returns and the trivial aspect `empty` shown in Figure 5 that simply calls back to this function. We compare the cost of calling `trivial`, when this function is not declared to be `hookable`, with the cost of calling `trivial` when it is declared to be `hookable` and has the aspect `empty` woven into its definition. We find that with the advice woven, invoking `trivial` costs 9.5 times as much as calling the function without the advice. In general, however, the cost of invoking advice varies with the complexity of the pointcut. Similar experiments have been performed for Java-based dynamic aspect systems and meta object systems. Table 6 summarizes the overhead of invoking dynamically woven advice as compared to a normal function call for several such systems (IguanaJ [29], MetaXa [22], Prose [28], and Guaranà [24]).

Tool		Ratio
μDyner		9.5
IguanaJ	(MOP)	24
MetaXa	(MOP)	28
Prose	(AOP)	40
Guaranà	(MOP)	70

Figure 6: Ratio between aspect invocation and ordinary call

The overall effect of the overhead of calling an aspect on the performance of an application depends on how often the

aspect is invoked. We again test the `quicksort` implementaion of Appendix A, with `partition` declared as `hookable`. In sorting an array of 10 million random integers, the trivial aspect of Figure 7 adds an overhead of less than 10% (8% in many cases) as compared to the original implementation of `quicksort` (using neither the aspect nor any `hookable` annotations).

```
empty :[
    int * partition(int * p, int * r) :[
            {
              return continue_partition(p,r);
            }
    ]
]
```

Figure 7: Benchmarking aspect.

5. RELATED WORK

μDyner is related to systems that allow modification of binary code, and to other aspect systems.

Several tools allow the rewriting of an executable after compilation and, in some cases, after linking. While this approach is relatively common in Java [5, 10, 13, 20, 32], it is relatively rare on native executables [3, 23, 30]. Eel [23] and Eth [30] can aggressively restructure the program, and can thus improve its performance. They cannot, however, modify a running executable and they offer APIs that are close to assembly language.

One tool that does allow dynamic instrumentation of a running native program. is Dyninst [3], which is based on the Unix debugging API (`ptrace`). This API is architectured in terms of the interaction between two processes, the process being debugged and the debugger. Dyninst instantiates the host process as the process being debugged and the process injecting new code as the debugger. This approach has high cost because `ptrace` requires the process being debugged and the debugger to synchronize on each written instruction. An instrumentation analogous to the weaving of the null aspect of Figure 5 requires 1.2s using Dyninst. Moreover, although the API is close to the C language, it seems difficult to trigger an advice execution on an access to a variable with Dyninst: the translation from the variable identifier to effective address is left to the user.

Cowan *et al.* [11] present an approach to dynamically loading and unloading new implementations of existing operations into running programs. Here the goal is to improve performance based on transitory invariants, rather than to add new functionality. They address the problem of incorrect interactions with outdated and incompletely installed code by using locks to ensure that no thread is executing relevant code during the replacement process. When there is indeed no process currently executing code that is scheduled to be replaced, they report an overhead of a few hundred cycles on a HP 9000 series 800 G70 (9000/887) dual-processor server. We are currently investigating efficient approaches to address this issue in the setting of μDyner.

AspectJ [21], targeted toward Java, is the de-facto lingua franca of all aspect systems. It offers a rich set of pointcut operators, but provides only weaving at compile time. AspectC [7] and AspectC++ [31] extend C and C++, respectively, with aspects. These systems also provide only static weaving. Douence *et al.* [15, 16] propose a formal model to define the semantics of aspect systems. We plan to try to merge the μDyner aspect model with this model, to take advantage of the ability to formally prove properties of the interactions between a collection of aspects.

To the best of our knowledge, no current aspect system provides all of the features of μDyner. In particular, most existing aspect systems provide only static, rather than dynamic, weaving.

6. CONCLUSION

Given the high proportion of HTTP traffic in the Internet, caching of documents is an important technique to reduce latency, bandwidth consumption, and server load. Augmenting a Web cache with a prefetching policy further improves its effectiveness. Nevertheless, there is no single prefetching policy that is best for all situations; good performance requires using a prefetching policy targeted to characteristics of the Web server and of the client. Thus, a Web cache should be extensible and provide the ability to load and unload new policies at run time.

In this paper, we have presented a new approach to implementing prefetching policies in an existing Web cache. We have shown that prefetching crosscuts the structure of a Web cache, and that it can thus appropriately be implemented as a collection of aspects. To address the need to install new policies in response to changing conditions, we have developed an architecture allowing the weaving and deweaving of aspects in an executing program. Our approach is based on the C language, which is typically used in the implementation of Web caches. Weaving and deweaving of an aspect have a relatively low cost, which is within the delay tolerated by TCP/IP retransmission mechanisms.

Our initial experiments with μDyner have shown that the system provides both good performance and a useful degree of expressiveness. In the short term, we are planning to use μDyner to implement a prefetching policy such as Interactive Prefetching (see Section 2.3) in Squid. In this context, we will investigate how a Web cache can be extended to itself download and deploy new prefetching policies. More generally, we are investigating whether these ideas can be used as a basis for a more general framework for constructing adaptable applications with critical performance and continuous service requirements such as operating systems.

7. REFERENCES

[1] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 263–274, Cambridge, MA, Aug. 1999.

[2] Apache Software Foundation. Apache. `www.apache.org`.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] X. Chen and X. Zhang. Coordinated data prefetching by utilizing reference information at both proxy and web servers. In *Proceedings of the 2nd ACM Workshop on Performance and Architecture of Web Servers, (PAWS-2001)*, June 2001.

[5] S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000 - Object-Oriented Programming, 14th European Conference*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336, Sophia Antipolis and Cannes, France, June 2000.

[6] K.-I. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[7] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Communications of the ACM*, 44(10):79–82, 2001.

[8] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing web latency. In *INFOCOM*, volume 2, pages 854–863, Tel Aviv, Israel, Mar. 2000.

[9] E. Cohen, B. Krishnamurthy, and J. Rexford. Efficient algorithms for predicting requests to web servers. In *INFOCOM*, volume 1, pages 284–293, New York, NY, Mar. 1999.

[10] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[11] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'96)*, pages 108–115, Annapolis MD, May 1996.

[12] D. Wessels and contributors. Squid web proxy cache. `www.squid-cache.org`.

[13] M. Dahm. Byte code engineering. In *JIT'99, Java-Informations-Tage*, pages 267–277. Springer, Sept. 1999.

[14] M. D. Dikaiakos and A. Stassopoulou. Content-selection strategies for the periodic prefetching of WWW resources via satellite. *Computer Communications*, 24(1):93–104, 2001.

[15] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, Pittsburgh, PA, Oct. 2002.

[16] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, Sept. 2001. Springer Verlag.

[17] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Evolutionary Trends of the Internet, Thyrrhenian International Workshop on Digital Communications, IWDC 2001*, volume 2170 of *Lecture Notes in Computer Science*, pages 556–575, Taormina, Italy, Sept. 2001.

[18] V. Issarny, M. Banâtre, B. Charpiot, and J.-M. Menaud. Quality of service and electronic newspaper: The Etel solution. *Lecture Notes in Computer Science*, 1752:472–496, 2000.

[19] Q. Jacobson and P. Cao. Potential and limits of Web prefetching between low-bandwidth clients and proxies. In *Proceedings of the Third International WWW Caching Workshop*, 1998.

[20] R. Keller and U. Hölzle. Binary component adaptation. In *ECCOP'98 - Object-Oriented Programming, 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329, Brussels, Belgium, July 1998.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[22] J. Kleinoder and M. Golm. MetaJava: An efficient run-time meta architecture for Java. In *Proceedings of the International Workshop on Object-Orientation in Operation Systems - IWOOOS '96*, pages 54–61, Seattle, Washington, Oct. 1996.

[23] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, CA, June 1995. ACM Press.

[24] A. Oliva and L. E. Buzato. The implementation of Guaranà on Java. Technical Report IC-98-32, Institute of Computing, University of Campinas, Campinas, Brazil, Sept. 1998.

[25] V. N. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implications. In *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 111–123, Stockholm, Sweden, Aug. 2000.

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

[27] V. Paxson and M. Allman. RFC 9188: Computing TCP's retransmission timer, Nov. 2000.

[28] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, Enschede, The Netherlands, Apr. 2002. ACM Press.

[29] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 205–230, Malaga, Spain, June 2002.

[30] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–8, Seattle, Washington, Aug. 1997.

[31] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, Feb. 2002.

[32] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 283–298, Pittsburgh, PA, Oct. 2002.

[33] Z. Wang and J. Crowcroft. Prefetching in world wide web. In *Proceeedings of Global Internet*, pages 28–32, London, England, Nov. 1996. IEEE.

[34] D. Wessels and K. Claffy. RFC 2186: Internet Cache Protocol (ICP), version 2, Sept. 1997.

[35] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, Kiawah Island Resort, SC, Dec. 1999.

[36] A. Yoshida. MOWS: distributed web and cache server in Java. *Computer Networks and ISDN Systems*, 29(8-13):965–975, 1997.

APPENDIX

A. ANNOTED QUICKSORT.C

```
inline void swap(int *i, int *j) {
  int temp;
  temp = *i;
  *i = *j;
  *j = temp;
}

int *partition(int *p, int *r) {
  int x, *i, *j;
  x = *p;
  i = p - 1;
  j = r + 1;
  while(1) {
    do { i++; } while(*i > x);
    do { j--; } while(*j < x);
    if (i < j)
      swap(i, j);
    else
      return j ;
  }
}

void sort(int *p, int *r) {
  int *q;
  if (p < r) {
    q=partition(p,r);
    sort(p,q-1);
    sort(q+1,r);
  }
}
```