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 enhance these benefits. For the best performance, however, the prefetching policy must match user and Web application characteristics. Thus, new prefetching policies must be loaded dynamically as needs change.

Most Web caches are large C programs, and thus adding one or more prefetching policies to an existing Web cache is a daunting task. The main problem is that prefetching concerns crosscut the cache structure. Aspect-oriented programming is a natural technique to address this issue. Nevertheless, existing approaches either do not provide dynamic weaving, incur a high overhead for invocation of dynamically loaded code, or do not target C applications. In this paper we present μ-Dyner, which addresses these issues. In particular, μ-Dyner provides a low overhead for aspect invocation, that meets the performance needs of Web caches.

Keywords

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

1. INTRODUCTION

Because HTTP amounts to over 80% of Internet traffic [17], caching HTTP documents is an appealing approach to reduce Internet latency and network bandwidth consumption [41]. Two factors, however, decrease Web cache effectiveness: (i) between 35% to 50% of Web documents are uncacheable because their content is specific to the initial request [41], and (ii) once cached, many Web documents are never requested again [28].

A strategy to overcome these limitations is to prefetch Web documents so that they are already in the cache when first requested [21]. 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 application 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 application can achieve a prefetch prediction accuracy of up to 92% [20]. To support the use of such policies, a Web cache should be extensible and support dynamic loading and unloading of new policies.

A prefetching policy impacts many functionalities of a Web cache, such as the choice of how and when to retrieve new documents and the strategy for deleting documents when the cache is filled. Because cache implementations typically do not export fine-grained interfaces to these functionalities, it is difficult to implement a prefetching policy as *e.g.* a new module for a module-based cache such as Squid [12]. Aspect-oriented programming (AOP), however, is a programming technique that is specifically directed towards such cross-cutting concerns. 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 Web applications.
- Continuous servicing. Loading or unloading a new policy should not interrupt the treatment of client requests. Thus, 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 latency and bandwidth.

AOSD '03 Boston, MA

This paper

This paper describes the μDyner AOP infrastructure for writing and dynamically deploying aspects in a running C program 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 a Pentium 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 2.24 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. All but at most the writing of a few instructions is transparent to the application, and thus there is typically little or no 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 use of μDyner and Section 4 describes its implementation. Section 5 evaluates the cost of inserting a μDyner aspect. Section 6 presents related work. Section 7 concludes and describes future work.

2. WEB CACHES AND PREFETCHING

We first describe 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 and show that AOP conveniently addresses these issues.

2.1 Web cache architecture

The goals of a Web cache are to decrease the average document access time, to decrease the bandwidth consumption, and to decrease the workload on each server encountered. To achieve these goals, 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 immediately sends the document to the user. Otherwise, the cache forwards the request to the Web server, downloads the document to its local storage, and returns the document 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 [40]. When such a *cooperative cache* does not already possess the requested document, it first attempts to find the document 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.

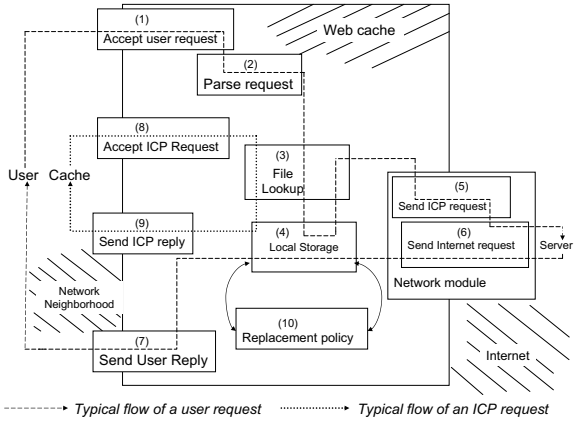

Figure 1: Structure of a modular Web cache

Web caches are often implemented as a collection of modules, each implementing a single concern, as illustrated in Figure 1 [12, 42]. 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 its neighbors (5) or to the Web server (6). If a new document is obtained, it is saved to the local storage (4), which may require activation of the replacement policy (10). Finally, the document is sent to the user (7). To communicate with a neighbor, the cache accepts and parses requests from the neighbor (8), 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 cache's storage space, 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 (1) or the ICP-request module (8) of the cache.
- The choice of documents to prefetch in response to a user request: Possible approaches include prefetching a few links near the top of the document [6] and statistics-based approaches that require the Web cache to maintain 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 cache's network module 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 chance of being accessed, and thus should not be the

highest priority candidates of the replacement policy. On the other hand, when a document is removed from the cache, the replacement policy should also remove documents prefetched for that document.

Possible approaches to adding prefetching to a Web cache include interposing a prefetching server in front of the Web cache, extending a modular Web cache with a prefetching module, and directly modifying the existing cache implementation. Interposition is transparent to the existing implementation, but allows little scope for interaction between the prefetching policy and the Web cache. For example, we have argued that prefetched documents should be treated differently by the replacement policy, implying a degree of interaction between the prefetching policy and the Web cache that goes beyond the interface typically exported by a Web cache to external servers. Dynamic loading of a new module implementing a prefetching policy, as allowed by Apache [2] and MOWS [42], simplifies communication between the prefetching policy and the rest of the cache, but this approach is still limited by the granularity of the interfaces provided by the existing modules. Modifying the cache implementation directly gives the prefetching policy access to the existing cache functions and data structures, but requires sophisticated knowledge of the cache implementation and is not well suited for the dynamic insertion and removal of prefetching policies.

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. 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 also assume a LRU replacement policy. To describe the aspects needed to implement this policy, we use the following AOP notions [24]:

- 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 current document itself. Thus, prefetching can only be initiated after the document is received by the network module. 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 execution context 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 (3), 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 (1).

A LRU replacement policy uses 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 storage module (4) must be adapted with an aspect that identifies new prefetched documents and sets their access times accordingly. The file lookup module (3) must similarly be adapted to update the access times of the associated prefetched documents when a requested document is reaccessed from the cache. Finally, the replacement policy (10) itself should be adapted to remove the associated prefetched documents when the requested document is removed from the cache.

The implementation of the interactive prefetching policy illustrates typical problems that arise when adapting a Web cache. The many prefetching strategies that have been developed [4, 8, 9, 14, 29, 39] vary, however, in how they crosscut a Web cache implementation. For example, the strategy of Cohen et al. makes prefetching decisions based on statistical observations about the history of incoming requests [9]. 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 variant 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. This variant thus crosscuts the network module. Rather than prefetching documents, the Web cache can simply preestablish a connection with the Web server [8], thus reducing the latency perceived by the user. Such a strategy crosscuts the network module, but differently than the use of a piggybacking protocol.

3. USING μDYNER

The μDyner aspect system provides the ability to dynamically weave and deweave aspects in a running C application. Three types of users interact with μDyner: the maintainer of the base program, the aspect developer, and the cache administrator. These roles are illustrated in Figure 2. The maintainer of the base program annotates the cache implementation to indicate where adaptation is allowed. The aspect developer identifies appropriate prefetching algorithms, writes the corresponding aspects, and compiles these aspects using μDyner. Finally, the cache administrator installs new prefetching policies as needs change.

3.1 Base-program annotation

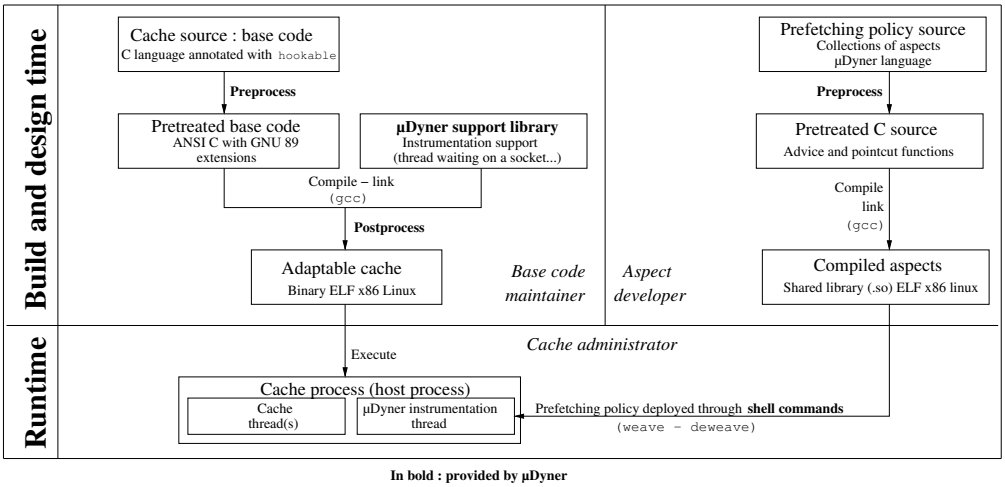

Figure 2: Processing to support dynamic weaving.

μDyner relies on a source instrumentation of the program to reduce the cost of dynamic weaving. Specifically, μDyner provides a source-level annotation `hookable` with which the base-program maintainer annotates the points in the program at which adaptation is allowed. Only functions and global variables (those that are declared at top level and are not declared to be `static`) can be declared to be `hookable`.

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

3.2 Aspect development

The aspect language syntax is defined in Figure 3. An aspect library contains a collection of aspects. The definition of each aspect consists of its name and a nested sequence of pointcut predicates that describe the affected join point. Each pointcut predicate but the last must describe a function call (`function-call`), and the sequence represents a sequence of direct nested calls. The innermost pointcut predicate can either be another function call 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-call`, or anywhere in the program if no `function-call` is mentioned. The sequence of pointcut predicates 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 pointcut. Execution of the advice replaces execution of the join point represented by the innermost pointcut predicate. Advice runs in the global address space of 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 predicate. The advice can furthermore invoke the original body of the function using the implicitly generated function pointer `continue_<function_name>`. A pointcut predicate representing an assignment (`global-variable-write`) includes both the name of

```
<aspect-library> ::= <aspect> | <aspect> <aspect-library>

<aspect> ::= <identifier> ":[" <pointcut-advice> "]"

<pointcut-advice> ::= <function-call> ":[" <pointcut-advice> "]"
                    | <function-call> ":[" <advice>"]"
                    | <global-variable-access> ":[" <advice> "]"

<function-call> ::= <type> <identifier> "(" <params> ")"

<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 3: 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 4: An extract of a prefetching policy.

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, which is responsible for performing the assignment, if desired. If the advice replaces a function call or variable reference, its return value is returned to the base program as the result of the join point execution. An advice that replaces a variable update should have no return value.

Figure 4 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 base program need only declare the function `relay_request` to be `hookable`.

3.3 Aspect deployment

μDyner provides the commands `weave` and `deweave` to deploy and undeploy aspects, respectively. The command:

```
weave <pid> <aspect-library>
```

weaves the compiled aspect library `aspect-library` into the process identified by `pid`. When only `pid` is given, `weave` lists the names of all of the aspects currently woven into the process `<pid>`. The command:

```
deweave <pid> <aspect-library>
```

deweaves `aspect-library` from the process `pid`.

4. IMPLEMENTATION OF μDYNER

We first present the run-time aspect structure of μDyner and then describe the compile-time and run-time processing that supports this structure. Because much of the weaving process is prepared at compile time, including processing of the source program and processing of the aspects, dynamic weaving has a low run-time overhead. Our current implementation is targeted towards the Pentium architecture. This implementation furthermore only allows at most one aspect per join point.

4.1 μ-Dyner run-time aspect structure

The goal of dynamic weaving is to connect the statically loaded base program to a dynamically loaded aspect. Because there can be many join points associated with a single aspect (in the case of an aspect that adapts a variable access), we implement this connection using a *hook*, which provides a level of indirection between the application and the aspect.

The basic form of a hook is as follows:

```
save registers
if (aspect_loaded && pointcut()) {
  call aspect
  restore registers
  return aspect result
} else {
  restore registers
  perform the original effect of the join point
  return result
}
```

The hook first checks a global variable `aspect_loaded` associated with the corresponding aspect library to determine whether weaving of the library is complete and then calls a function `pointcut` that checks whether current call stack matches the pointcut of the aspect. If the pointcut has only one predicate, the execution context of the join point is trivially satisfied and the call to `pointcut()` is omitted. If both tests are satisfied, then the aspect is invoked. Otherwise, the hook performs the effect of the join point before returning to the application. Because a hook contains information about both the base program and the aspect, hooks are created dynamically, during weaving. We must thus explicitly include the saving and the restore of registers that are normally introduced by a compiler before calling a function. Furthermore, because a hook contains information about the address and contents of each join point, a separate hook is created for every join point.

Figure 5 shows two hookable functions, `handle_request` and `relay_request`. No aspect has yet been loaded that adapts `relay_request` so there is no hook associated with this function. The aspect `prevent_propagation` adapts the function `relay_request`. This function thus begins by jumping to a hook that tests the pointcut of `prevent_propagation` and possibly jumps to the associated advice.

4.2 Compile-time processing of the base program

Weaving is performed directly on the executable image of the base program, which is already loaded into memory. This process requires the ability to identify join points in the executable code and to modify these join points to jump to the advice. μDyner modifies the declarations of `hookable` functions and global variables to ensure these capabilities. As illustrated in Figure 2, the overall compile-time processing consists of modification of the source program by the μDyner preprocessor, compilation of the resulting program using `gcc`, and modification of the generated code by the μDyner postprocessor.

A function-call join point is implemented by a jump to the hook at the beginning of the function definition rather than at each of the call sites. The `hookable` annotation causes the μDyner preprocessor to modify the function such that there is sufficient space (5 bytes for the Pentium) at the beginning of the function to insert this jump. In Figure 5, the definition of `handle_request` illustrates the coding of a hookable function before the weaving of an aspect, and the definition of `relay_request` illustrates the coding of a hookable function after weaving. The μDyner preprocessor also modifies the source program to prevent inlining of non-leaf functions and all hookable functions, thus making it possible to find the starting address of each hookable function in the compiled code and to detect the presence on the call stack of functions potentially mentioned by pointcuts.

A variable-access join point is implemented by inserting a jump to a hook at each access to the variable. 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 macro-expands into a `volatile` declaration, which forces the C compiler to implement every access to the variable as an explicit access to the associated memory location.[1] On the Pentium, an access to a memory location occupies at least as many bytes as a jump to an explicit address, and thus no extra space is reserved.

After this preprocessing, the resulting program is com-

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

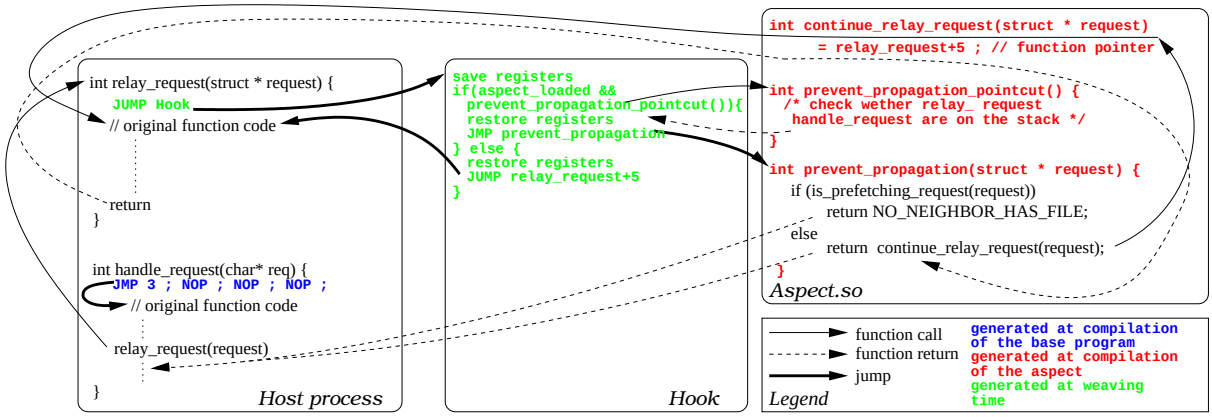

Figure 5: Execution of an aspect.

piled using `gcc`.[2] The μDyner postprocessor then reorganizes the beginning of each hookable function such that the 5 `NOP`s inserted by the μDyner preprocessor precede the function prelude inserted by the C compiler, and overwrites the first two of these `NOP`s with a short jump to the first instruction of this prelude. After this modification, the instructions following the `NOP` instructions represent a complete, standard function definition, that can be invoked in the normal way.

4.3 Compile-time processing of the aspect

For each aspect, the μDyner compiler creates a function checking the pointcut and a function implementing the advice. This code is generated as C code. The pointcut and advice functions associated with all of the aspects in an aspect library are then compiled together as a library using a standard C compiler.

The structure of an advice function depends on the innermost pointcut predicate of the aspect. When the innermost pointcut predicate is a function call, the advice function is parameterized by the arguments of the called function. When the innermost pointcut predicate is a global variable reference, the μDyner compiler generates an external global declaration for the variable, and thus the advice function can refer to this variable directly. When the innermost pointcut predicate is a global variable update, the affected global variable can again be accessed directly. The value of the right-hand side expression is obtained from a parameter named as indicated by the right-hand side of the description of the pointcut predicate. The return type of the advice function is the return type of the construct indicated by the innermost pointcut predicate.

4.4 Weaving

Weaving at run-time loads the requested aspect library into the address space of the host process using `dlopen`, and creates hooks to link the join points in the application to the corresponding aspects. The manner in which this linking is carried out and the specific contents of a hook depend on the kind of join point being adapted. Generally, we avoid freezing the application during the weaving process.

[2]The use of `gcc` is only required because we express the inserted assembler code using `gcc` syntax.

For an aspect whose innermost pointcut predicate is a function call, the jump to the hook is placed in the reserved space at the beginning of the function definition. Because such a jump requires 5 bytes and the Pentium only provides 2-byte and 4-byte atomic memory write operations, such a jump cannot be inserted atomically. Thus, the weaving process first installs the second half of the jump to the hook behind the short jump to the beginning of the function body and then overwrites this short jump with the first half of the jump to the hook. This approach requires no freezing of the host process. Furthermore, any write to memory automatically invalidates corresponding cache information, so the cache remains coherent [19].

A hook for an aspect whose innermost pointcut predicate is a function call has the following form, where `advice_fn` is the address of the advice function and `fn` is the address of the function mentioned in the pointcut predicate:

```
save registers used by the hook
if (aspect_loaded && pointcut())  {
  restore registers used by the hook
  JMP advice_fn
} else {
  restore registers used by the hook
  JMP fn + 5
}
```

If the pointcut is satisfied, the hook jumps to the advice function. The use of a jump has no effect on the stack and thus leaves the arguments and return pointer at the top of the stack as they were placed there by the original function call. Thus, the advice can freely refer to the original arguments via its parameters, and a return instruction in the advice will cause a return all the way back to the original call site. If the pointcut is not satisfied, the hook jumps to the first instruction in the original definition of the function. The use of a jump instruction again ensures that the original function sees the original arguments and return pointer at the top of the stack. A hook for a function-call join point need only save and restore the callee-save registers that it uses; other registers required by the calling context have been saved at the call site. Weaving of an aspect whose innermost pointcut predicate is a function call also initializes the function pointer `continue_<function_name>` with the address of the first instruction representing code from the original function definition.

For an aspect whose innermost pointcut predicate is a variable access, it seems impossible to complete the normal semantics of the instruction while overwriting the instruction with a jump to the hook. We thus first overwrite the first byte of the variable reference instruction with a `INT3` instruction, which interrupts the process. The associated signal handler adjusts the return address of the process to the beginning of the join point itself and then returns, thus placing the process in a loop until the insertion of the jump to the hook is complete. As in the case of the weaving of a function-call join point, we then write the second half of the jump instruction behind the `INT3` instruction, and finally overwrite the `INT3` instruction with the first half of the jump. At this point, if the process is looping, it jumps to the hook. Weaving at a variable-access join point can thus stop the application, but only if the application is executing code affected by the weaving process, and only for the amount of time required for a few memory writes.

A hook for an aspect whose innermost pointcut predicate is a variable reference has the following form, where `vaddr` is the address of the referenced variable, `addr` is the address of the join point, $\mathtt{r}_{\mathtt{tgt}}$ is the target register of the instruction at this address, and `eax` contains the return value of the advice:

```
save registers
if (aspect_loaded && pointcut()) {
  CALL advice_fn
  MOVE eax, rtgt
  restore registers except rtgt
} else {
  restore registers
  MOVE vaddr, rtgt
}
JMP addr + move_instruction_size
```

If the pointcut is satisfied, the hook calls the advice and stores the return value in the target register of the original variable reference instruction. Otherwise, the hook performs the original variable-reference instruction. In either case, the hook then jumps back to just past the join point. The hook must save and restore the registers that it uses as well as all of the callee save registers; unlike the case of a function-call join point, the context of a variable-reference join point does not prepare the registers for an eventual function call.

A hook for an aspect whose innermost pointcut predicate is a variable update has the following form, where `vaddr` is the address of the affected variable, `addr` is the address of the join point, $\mathtt{r}_{\mathtt{src}}$ is the register containing the value of the right-hand side expression of the assignment as determined from this instruction :

```
save registers
if (aspect_loaded && pointcut()) {
  PUSH rsrc
  CALL advice_fn
  restore registers
  MOVE vaddr, rsrc
} else {
  restore registers
  MOVE rsrc, vaddr
}
JMP addr + move_instruction_size
```

If the pointcut is satisfied, the hook first pushes the value of the right-hand side expression of the assignment on the stack as the argument to the advice. It then calls the advice, which is responsible for performing the assignment if desired. Finally, the hook stores the value of the affected variable into the source register of the assignment, in case the application uses this register to obtain the value of the assignment statement. If the pointcut is not satisfied, the hook performs the original update operation. In either case, the hook then jumps back to just past the join point. The saving and restoring of registers for a variable-update join point is done in the same manner as for a variable-reference join point. This hook illustrates the case where the value of the right-hand side expression of the update is in a register. The case where this value is a constant is similar.

When the weaving of an entire aspect library is complete, the weaver sets the corresponding `aspect_loaded` variable, allowing the woven advice to be executed.

4.5 Deweaving

Deweaving an aspect restores the code at each join point, frees the space allocated for the hook, and unloads the aspect library using `dlclose`.

5. 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 3.2.1 with the option `-O2`). As a test program, we use a version of the TinyProxy Web proxy (version 1.0 [22]) in which we have added a hash table to provide caching.

The `hookable` annotation introduces a short jump 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 introduced overhead, we compared the performance of our modified version of TinyProxy with and without `hookable` annotations on all functions and global variables. We measured the duration between the point in the execution where TinyProxy received the complete from the client and the point where Tinyproxy started to send the answer to the client on a variety of miss and hit cases. This macro benchmark shows the introduced overhead by the `hookable` annotation as compared to using ordinary functions and global variables was not measurable.

Weaving an aspect library into the base program requires loading the aspect and instrumenting the affected join points. The cost of loading the aspect library is dominated by the cost of `dlopen`; loading the aspect library of Figure 6 into TinyProxy requires 199 μs out of a total weaving cost of 266 μs. The instrumentation process creates a hook for each join point and overwrites the join point with a jump to this hook. Creating a hook requires allocating space for the code (using `malloc`) and instantiating this space with the hook instructions. The cost of creating a hook is about 10 μs. Updating a function-call join point with a jump to the hook amounts to a write of a few instructions and has negligible cost. Updating a variable access join point with a jump to the hook can require freezing the application using the `INT3` instruction. Because this instruction involves a system call, installing such a jump can require up to 25 μs. Nevertheless, `INT3` is only executed if the application is trying to execute the variable access concurrently with the weaving, which should occur only rarely. In the most common case, the cost of installing the jump is identical to that of a

function-call join point. For a variable access join point, 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. The remainder of the total cost of loading an aspect library is for symbol resolution.

Deweaving inverses the process of weaving. Each of the operations involved is equally or less expensive than its weaving counterparts. For example, unloading the aspect library of Figure 6 with `dlclose` requires only 119 μs. Thus, the cost of deweaving is lower than the cost of weaving.

To assess the effect of weaving on the cache's ability to treat incoming requests, we compare the potential application freeze time induced by weaving to the TCP retransmission timeout [31]. Typical values on a BSD implementation of TCP vary between 0.5 and 1 second [1]. Furthermore, it has been argued that the timeout should not be lower than 250ms [1]. Given the above analysis of the potential freeze time per join point, even if the application is frozen on the instrumentation of every join point, we can still weave over 14,000 join points within 250ms.

```
empty :[
    int get(char *url) :[
        { return continue_get(url); }
    ]
]
```

Figure 6: A null aspect.

To assess the overhead of calling a function adapted by an aspect, we consider a modified version of `get` that returns immediately and the null aspect of Figure 6 that simply calls back to this function. We compare the cost of calling `get` when it is not declared to be `hookable`, with the cost of calling `get` when it is declared to be `hookable` and has the aspect `empty` woven into its definition. With the advice woven, invoking `get` costs 2.24 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 7 summarizes the overhead of invoking dynamically woven advice as compared to a normal function call for several such systems (IguanaJ [34], MetaXa [25], Prose [33], and Guaranà [27]). These approaches introduce a much higher overhead than μDyner. MOP-based approaches require reifying the entire method call expression. Prose implements a join point as a break point that passes control to a separate debugging process that determines whether to trigger execution of the advice.

Tool		Ratio
μDyner		2.24
IguanaJ	(MOP)	24
MetaXa	(MOP)	28
Prose	(AOP)	40
Guaranà	(MOP)	70

Figure 7: 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 compare the performance of the `get` function of our modified TinyProxy adapted by the aspect of Figure 6 to this function with neither the aspect nor any `hookable` annotations. We measure an overhead of under 1.5%, but this is well within the variation in the performance of `get` itself, which is up to 20%.

6. 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, 23, 38], it is relatively rare on native executables [3, 26, 35].

Dyninst [3, 18] allows instrumentation of a running native program. This tool relies on the Unix debugging API (`ptrace`) that involves 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. 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 an effective address is left to the user. Finally, Dyninst relocates the instrumented instructions into trampolines. This is unsafe when these instructions are the target of jump instructions. Dyninst addresses this issue using heuristics run at instrumentation time [18].

Tamches and Miller [37] have proposed a technology allowing fine grained dynamic instrumentation of RISC operating systems. They solve the problem of inserting a jump to dynamically loaded code by overwriting the existing instruction with a short jump to space following the function that is known to be empty due to kernel alignment constraints. They then place a longer jump in this space to the dynamically loaded code. This solution is not applicable to a user-level Pentium application, because such applications rarely if ever contain any free space between compiled functions.

Several approaches have used dynamic adaptation to improve the performances of Web caches by adapting the underlying machine. Tamches and Miller have used their technology to tune Solaris to Squid [37]. Piumarta et al. similarly tune a virtual machine to a bytecode compiled Web cache [32].

Cowan *et al.* [11] present an approach to dynamically loading and unloading new implementations of existing operations into running programs. Their 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. We have used the `aspect_loaded` flag to address this issue.

AspectJ [24], targeted towards 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++ [36] extend C and C++, respectively, with aspects. All three of these approaches provide only static weaving. JAC [30] is a dynamic weaver for Java relying on interposition objects introduced at load time. The use of such interposition objects can make the application more than 5 times slower [30]. Douence *et al.* [15, 16] propose a formal model to define the semantics of aspect systems. We plan 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 systems provide only static, rather than dynamic, weaving.

7. 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 application 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 *dynamically* 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 the implementation language of most existing 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 it provides good performance. Our immediate plans are to eliminate the need for interaction with the maintainer of the base program, by allowing adaptation of all functions and global variables, and the need for interaction with the cache administrator, by extending the Web cache to monitor incoming requests and select appropriate prefetching policies as conditions change. We are also working on allowing multiple aspects at a single join point. In the short term, we plan to use μDyner to implement a prefetching policy such as interactive prefetching (see Section 2.3) in Squid. 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.

8. REFERENCES

[1] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *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 *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 *Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[7] Y. Coady, G. Kiczales, J. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 79–86, St. Emilion, France, Sept. 2002.

[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 *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, 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 *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 *3rd International Conference on Reflection and Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186,

Kyoto, Japan, Sept. 2001.

[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] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE PACT*, pages 201–213, 1997.

[19] Intel Corportation. *IA-32 Intel Architecture Software Developer's Manual*, volume 3: System programming Guide. Intel Corportation, 2001. section 7.1.1 page 7-2.

[20] 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.

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

[22] R. J. Kaes and S. Young. Tinyproxy. `http://tinyproxy.sourceforge.net/`.

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

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

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

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

[27] 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.

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

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

[30] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac : A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[31] V. Paxson and M. Allman. RFC 2188: Computing TCP's retransmission timer, Nov. 2000.

[32] I. Piumarta, F. Ogel, C. Baillarguet, and B. Folliot. Applying the VVM kernel to flexible Web caches. In IEEE, editor, *The Eighth Workshop on Hot Topics in Operating Systems: [HotOS-VIII]: Schloss Elmau, RFA*, pages 178–178, 2001.

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

[34] 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.

[35] 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 *USENIX Windows NT Workshop*, pages 1–8, Seattle, WA, Aug. 1997.

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

[37] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[38] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *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.

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

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

[41] 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.

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