Aspectizing a Web Server for Adaptation

Wasif Gilani, Julio Sincero, Olaf Spinczyk
Friedrich-Alexander University Erlangen-Nuremberg
{gilani,sincero,spinczyk}@cs.fau.de

***Abstract*— Web servers are exposed to extremely changing runtime requirements. Going offline to adjust policies and configuration parameters in order to cope with such requirements is not an available choice for long running web servers. Many of the policies that need to be adapted are crosscutting in nature. Aspect-Oriented Programming (AOP) provides mechanisms to encapsulate the crosscutting policies as aspects. This paper describes the integration of a statically configurable web server with our dynamic aspect weaving infrastructure. This integration transformed the server to a dynamically adaptable one that could adjust its policies and configuration parameters at runtime according to the changing requirements. This paper further provides a comprehensive analysis of the memory and runtime costs associated with this transformation, and explains how our dynamic aspect weaving infrastructure via its tailored support facilitates to minimise these costs.**

I. Introduction

A web server is an application that accepts HTTP requests from some client and provides an HTTP response which is typically an HTML document. The traffic to a web server is highly unpredictable and varies from extreme low to very peaky which could be orders of magnitude greater than the average. A very slow or overloaded server is under risk of losing, driving away customers due to unsatisfactory performance. This is especially true for trade or e-commerce servers, which should produce a response to the clients within a sufficiently small amount of time, or otherwise face loss of customers and thereby, revenue.

The two traditionally employed techniques to deal with the extremely variant loads are, either to employ redundant hardware, or to statically configure different policies and configuration parameters. In the case of redundant hardware, enough web server machines are provided to handle the peak loads that the site could experience. However, this approach completely ignores the cost issues which arise when scaling a site to a large farm of machines. In the second approach, the web servers adjust different configuration parameters, such as the session time-out value, maximum number of threads , etc., and policies, such as concurrency, load balancing, caching, security, scheduling, etc., to offer optimize performances. Most adjustments are performed statically, and are typically not driven by the monitoring and feedback of system performance. However, because of the inherent dynamic nature of the Internet, it is simply not realistic to determine statically the different configuration limits. A policy or configuration parameter, which is configured statically, may be appropriate at one point, but may not be valid later, and the system may not gracefully handle the new requirements. The setting of inconsistent limits could result in either under-utilization or over-utilization of servers.

Many of the policies and strategies, such as synchronization, security, profiling, tracing, scheduling, etc., are crosscutting in nature. With traditional object-oriented techniques, the adaptation of such policies generally requires a system-wide change, and often a major redesign of the system. Aspect-oriented programming (AOP) addresses the problem of crosscutting by providing mechanisms to separate the crosscutting concerns as aspects. The “aspectization” process leads to a highly modularized, and fine-grained system, where the crosscutting policies and strategies, encapsulated as aspects, are allowed to adapt and evolve in isolation without affecting the rest of the system.

In this paper, we describe our experience with the transformation process of a statically configurable web server to a dynamically configurable one by integrating it with our dynamic aspect weaving infrastructure. This paper provides details and results about how our weaving infrastructure allows to tailor the adaptation costs according to the requirements and available resources. The remaining paper is organized as follows. We start with the motivation. This is followed by a brief description of our dynamic aspect weaving infrastructure. Section IV presents the transformation process of a statically configurable web server to a dynamically adaptable one, and a comprehensive analysis of the associated memory and runtime costs. Finally, section VI concludes the paper.

II. Motivation

Because of the various limitations with static configuration, there is a strong requirement for web servers to be able to dynamically adapt according to the runtime changes such as the network and client requirements, etc. The dynamic adaptation of a web server means switching at runtime to most appropriate policies such as concurrency (threading strategies), scheduling, connection management, load balancing, synchronization, etc., as well as the adjustment of different configuration parameters such as time-out, maximum number of threads, simultaneous connections etc. The ability to swiftly adapt according to the current workload is a promising approach for the commercial success of web sites as it offers a number of benefits such as maximizing the throughput, and the reduction of response times, etc. Therefore, instead of developing web servers on the basis of the static knowledge of the resources and various load conditions, it is much more efficient and economical to make use of the current characteristics, such as the request load, number of simultaneous connections, type of

request, requested file size, etc., to automatically tune various policies and configuration parameters dynamically.

The key to developing highly adaptable and efficient web servers is through a design which is flexible enough to accommodate different policies for dealing with varying server load and the type of incoming requests at runtime. The application of AOP helps to isolate and cleanly encapsulate the crosscutting policies into aspects. The dynamic AOP allows the policies encapsulated as aspects to be added and removed at runtime according to the load statistics resulting in a highly adaptable server. Even the configuration parameters could be handled by means of dynamic aspects according to the load and the available resources.

The costs introduced due to dynamic aspect weaving infrastructure can be divided into two parts: the cost of the runtime system, and the cost of hooks. The fixed runtime system support and the absence of a filtration mechanism in many of the available dynamic aspect weavers unnecessarily increases the costs of dynamic aspect weaving. The hooking process is particularly memory demanding and the absence of a filtration mechanism leads to insertion of hooks at all join points in the target application, regardless of their relevance to the adaptation of any policy or strategy. Our dynamic aspect weaving infrastructure offers a feature-rich dynamic aspect weaving support that could be tailored according to the specific requirements and available resources, resulting in an extremely optimized dynamic adaptation support. The support for a powerful filtration mechanism further means that the hooks are inserted only at the relevant set of join points thereby effectively minimising the hooking overhead. The relevant set of join points correspond to specific locations in the code where the adaptations are anticipated to happen. The transformation process of the web server presented in this paper demonstrates that a very limited set of join points are actually relevant for the adaptation of various policies and strategies.

III. A Family-Based Dynamic Aspect Weaving Infrastructure

For the development of our dynamic weaver infrastructure, we had two objectives. First, to provide a feature-rich dynamic aspect weaver that could be tailored according to specific requirements and available resources, and second, to bring down the cost of dynamic weaving. We applied the software product line (SPL) approach to the dynamic aspect weaving domain and come up with the family-based weaver. The tailored weavers are generated by selecting only the required set of AOP features from the weaver family. A variant management system called *pure::variants* is employed to completely automate the weaver generation process [2]. The support for adaptation, provided by the weaver family, ranges from completely unanticipated to unanticipated. For handling unanticipated adaptations, the dynamic aspect weaver family offers the most comprehensive instrumentation support that helps to expose each and every location in the target application for adaptations. This in combination with a feature-rich runtime system enables to carry any type of adaptations in any location of the target application. For handling anticipated adaptations, the weaver family offers mechanisms, which vigorously exploit the *a-priori-knowledge* of the target application, to bring down the dynamic aspect weaving cost only due to actually affected joinpoints, actually woven aspects, and used AOP features [6]. The optimizations performed by the exploitation of *a-priori-knowledge* are comparable to the ones offered by static weavers, which basically exploit the same information for this purpose: actually affected joinpoints, aspects, and used AOP features. The main difference is that this information is implicitly available to static weavers, while it has to be explicitly provided for the generation of a tailored dynamic weaver.

The current implementation of the weaver family is carried out in C++, and employs the source code instrumentation approach [6]. The AspectC++ [13] static weaver is employed as a hooking platform in our family-based weaver. Though the weavers that support runtime hooking offer more efficient solution since hooks are inserted only at the join points affected by the applied adaptations, the employment of runtime hooking techniques compromises the portability of weavers, and render them either JVM-specific [12], [11], [3], or architecture-specific [15], [5], [4]. The source code instrumentation approach promises portability, and the availability of a powerful filtration mechanism via the AspectC++ pointcut mechanism means that the hooking costs of our weaving infrastructure can be effectively minimised by inserting hooks only at the relevant set of join points.

IV. MyServer Project

MyServer is an open source web server, and is implemented C++. It is freely available under gnu license, and supports many standards and protocols required to build up a web server. Currently, MyServer is configured statically at compile-time like most of the available web servers. In the current implementation, the concurrency policy is statically configured to be thread-pool. The number of threads in the thread-pool that remain alive throughout the runtime, and the maximum number of threads the server can create, are defined statically. Similarly, another configuration parameter time out, which handles time out for connections, is arbitrarily selected and is set at the value of 60 seconds by default. Clearly, the imposition of static resource limits and fixed policies, as is done in the MyServer project, do not correspond to the extremely unpredictable behavior of the Internet traffic, load, request characteristics, or the available resources, etc. Any inadequate static selection could result in performance degradation, reduced throughput, or under utilization of resources. The capability to dynamically reconfigure the configuration parameters and the policies according to the current load characteristics promises higher performance and throughput.

The transformation process of MyServer project, from static to a dynamic server, started with the integration of MyServer with our dynamic aspect weaving infrastructure. The idea was

```
aspect instrument{
 pointcut virtual dynamicJPS()=0;
  public:
   advice dynamicJPS():before(){
    monitor<advInfo,JoinPoint::JPID>::BeforeAdvice();
   }
};


aspect instrumentAll:public instrument{
 pointcut virtual dynamicJPS() = execution("% ...::%(...)")
                               ||call("% ...::%(...)");
};
```

Fig. 1. A static preparation aspect employed for instrumentation in MyServer

to isolate and encapsulate the various crosscutting policies as aspects, and even to control the various configuration parameters by embedding them into dynamic aspects. The dynamic aspects afterwards could be woven or unwoven from the server according to the emerging requirements.

A. Instrumentation Policy

The integration of MyServer with our dynamic aspect weaver began with the instrumentation of its source code. The instrumentation process is the most demanding one in terms of memory resources and runtime overhead. This case study started with the analysis of the costs associated with the different instrumentation approaches in order to demonstrate the importance of a filtered instrumentation support, which, as discussed before, is a missing feature in most of the available dynamic aspect weavers.

The source code of MyServer was instrumented with varying instrumentation policies in order to evaluate the overhead associated with each of the policies. All measurements were performed on a Pentium-4 M, 2.8 GHz machine running Linux kernel 2.4, and with a gcc 3.3.3 compiler. In the first case, all *call* and *execution* join points of the project were instrumented with *before* hooks. This means that at runtime, the dynamic weaver could execute adaptation code via dynamic aspects only *before* the *call* and *execution* of all hooked join points. The static preparation aspect employed for hooking is shown in Figure 1. It can be seen that each hooked join point is allocated a unique runtime monitor object by employing a template monitor class. The *instrument* aspect employs the unique numeric identifiers *(JointPoint::JPID())*, assigned to each hooked join point by the static AspectC++ weaver, for generating unique monitor objects. These numeric identifiers along with a range of static and dynamic context information is available via the join point API of the static AspectC++ weaver. The allocation of a unique runtime monitor object to each join point is different from the traditional approach in the dynamic weaving domain where a single centralized runtime monitor takes care of all interaction between the join points and aspects. The implementation based on a centralized monitor proved to be quite expensive since each time when a hooked join point is invoked in the control flow, the whole list of join points maintained by the single centralized monitor is traversed to find out the invoked join point. Therefore, even if there are just empty hooks with no advice registered, this model causes significant runtime overhead. The allocation of unique runtime monitor objects for each hooked join point means that the involved complexity for join point look-up is effectively reduced to *O(1)* in contrast to the *O(log N)* complexity of the centralized runtime monitor model, where N is the number of hooked join points. The monitor template class is also passed another parameter *advInfo*. The *advInfo* corresponds to the specifc advice type in the dynamic aspect implementation that has to be registered against the affected join point.

The join point project repository generated by the static AspectC++ weaver during the weaving process provides the total number of hooked join points, which was 3378 join points in this case when all *execution* and *call* join points were hooked. Out of this, the number of *execution* join points was 865, whereas the number of *call* join points was 2513. Both versions were compiled with the *Os* compiling option. The version, with all join points hooked, consumed a total of 399590 bytes of memory. The non-instrumented version consumed 300428 bytes of memory. This means that the full instrumentation policy consumed 1.3 times more memory. The difference in memory consumption between the instrumented and the non-instrumented version divided by the total number of instrumented join points gives an average cost of the hook. In the case of MyServer project, the cost turned out to be 29 bytes per join point, which is 12 bytes more that the actual cost of 17 bytes for a simple hook, as presented in [6]. The 17 byte cost was calculated for the test case where the functions were void. But, all join points, which are not void, and return some type, consume extra bytes according to their return type.

The analysis of the binary code further revealed that there was additional code for implementing stack unwind semantics in all hooked functions, which were not void, and were compiled with exceptions enabled. AspectC++ creates a result buffer for each hooked function even if the functions are returning primitive object types, for example int, etc. However, the overhead of such a result buffer makes sense only if the functions return some user defined object types. The g++ compiler fails to optimize and, therefore, throw unnecessary stack unwind code. Though, the employment of flags *-fno-exceptions, and -fno-rtti* helped to avoid this overhead, some files of the MyServer project that used exceptions had to be compiled with exceptions. It was further found out that the C++ compiler was not inlining large functions when they were hooked. This results in an extra call in large functions that are called only once, and, therefore, causes unnecessary performance overhead.

Apart from imposing varying memory overhead, the various hooking policies affected the server response time as well. To analyze the degradation in performance, in relation to the varying hooking policies, a number of test cases were carried out. For runtime measurements, MyServer was run on a dedicated Pentium-4 2.8GHz machine with 512 MB of RAM. The server operating system was Linux kernel 2.4.

```
aspect measureTime{
  char time_string[40];
  struct timeval tv1;
  struct timeval tv2;
  long end_time;
  int count;
  Time_entry time_table[10000];
  pointcut control () = execution ("int Http::
                                    controlConnection(...)");
public:
  measuretime() { count = 0; }
  advice control () : before () {
    gettimeofday(&tv1, NULL);
    localtime (&tv1.tv_sec);
  }
  advice control () : after () {
    gettimeofday(&tv2, NULL);
    localtime (&tv2.tv_sec);
    end_time = (tv2.tv_sec - tv1.tv_sec) *
               1000000 + ( tv2.tv_usec - tv1.tv_usec);
    time_table[count].e_time = end_time;
    count++;
  }
};
```

Fig. 2. An aspect used for measuring response times in MyServer

A benchmark called Httperf [10] was used for generating the client requests. It is a freely available benchmark, which supports both HTTP and HTTPS protocols. Httperf offers flexible mechanisms to generate a continuous flow of HTTP requests issued from one or more client machines. In order to avoid effects of network latencies on measurements, the benchmark was run on the same dedicated machine like that of MyServer. The generated clients accessed a static page hosted by MyServer. The server response time was measured with the help of a static aspect instead of employing the Httperf tool. This was due to the reason that each of the incoming requests spents some non-deterministic time in the MyServer queue. This time was irrelevant in order to determine the accurate impact of instrumentation on the server response time. The employment of a static aspect helped to exclude that time from the measurement, and helped to produce very accurate results. Nothing needed to be changed in the source code of MyServer except weaving of a *measureTime* aspect, as shown in Figure 2. The *measureTime* aspect measured the time elapsed from the instance the request was handed over to a free thread till the request was serviced.

In the first test case, the original version of MyServer without instrumentation was run. A workload of 10000 requests, at the rate of 10 requests/sec, was generated with Httperf, and the server response time was measured. Afterwards, a version of MyServer with full instrumentation was run, and the response time was measured for the same load. Figure 3 shows the response time for the instrumented and non-instrumented versions. It can be seen from graphs that the full instrumentation policy resulted in comparatively higher response times. The increase in response times is due to the additional runtime checks incurred due to hooked join points that are executed when a client request is served. This showed that the full instrumentation policy leads to a comparatively higher memory and runtime overhead.

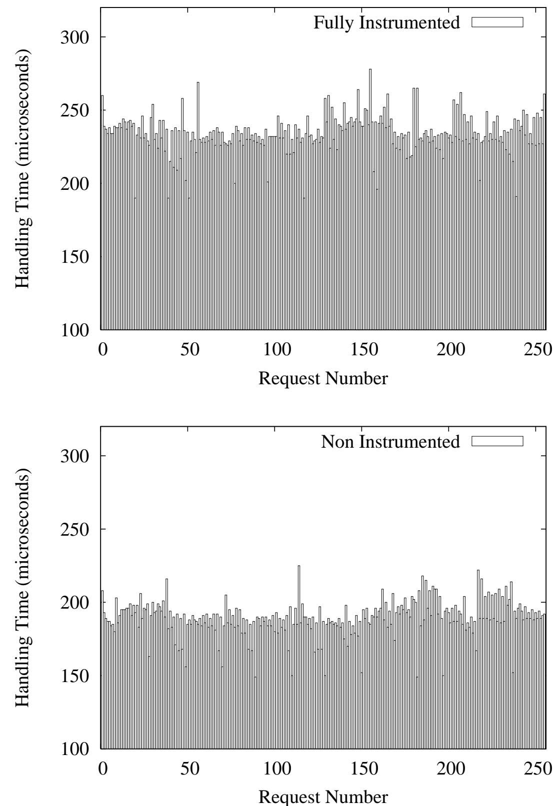

Fig. 3. Response time measurement of fully instrumented and non-instrumented versions of Myserver

MyServer is a simple and small webserver, where the total number of join points is not that large. Therefore, even with full instrumentation policy, the total size of the system was approximately 399 kilobytes of memory. This much cost is affordable in some instances in return for the support for unanticipated adaptations since all locations are exposed for adaptations. But, if one has to deal with larger systems, like middleware, or operating systems, etc., the total number of join points could be phenomenal. A full instrumentation, therefore, would be an ill-advised policy even for larger systems.

A good percentage of join points in a system are irrelevant for the adaptation of policies. A closer look into the join point information repository of MyServer, generated by AspectC++, revealed that many of the hooked join points carried no relevance for adaptation. Table I shows some of the irrelevant join points in the Myserver source code, and the number of their occurrences. There were many more irrelevant join points, like the ones shown in the table, and the unnecessary hooking of such join points serve no purpose other than imposing memory and runtime overhead. The dynamic aspect weaver family, via its join point filtration mechanism, allows to exclude all irrelevant join points during the hooking process. In the case of MyServer project, a pointcut description shown in Figure 4 resulted in the exclusion of all functions of the standard library, which don't generally play any role in the adaptation

Join Points	Total Occurrences in the source code
strcpy(...)	77
strncpy(...)	18
strcmp(...)	24
strncmp(...)	4
strupr(...)	4
strstr(...)	7
strcat(...)	38
strtok(...)	36
strcasecmp(...)	86

TABLE I

IRRELEVANT JOIN POINTS IN TERMS OF THE ADAPTATION OF POLICIES IN MYSERVER.

```
pointcut std_function_calls() = call("% std::%(...)");
pointcut virtual dynamicJPS() = execution("% ...::%(...)")
         || call("% ...::%(...)")&& !std_function_calls();
```

Fig. 4. A pointcut to exclude calls to standard library functions from the hooking process.

or evolution of systems policies. The pointcut resulted in the exclusion of around 294 unnecessary join points from the instrumentation process. This meant straight away saving of 8526 bytes of memory that was unnecessarily consumed due to the hooking of standard library functions. Still, there were a large number of join points in MyServer, which were hooked during the instrumentation process, and which had no active role to play in the adaptation or evolution of the server.

B. Aspectizing MyServer

A closer look into the source code of MyServer revealed that the implementation code for many policies and strategies like synchronization, concurrency, logging, etc., was scattered across multiple functions. The code implementing the policies was tightly coupled with the functional code, and, therefore, their adaptation was extremely complex, and would have required a system-wide change. The crosscutting further made the systems basic mechanisms too difficult to understand and maintain because of the complications due to policy specifications in them. AOP was applied as a solution to isolate and cleanly encapsulate the crosscutting policies into reusable aspects. Dynamic weaving was then employed for the adaptation and evolution of policies according to the changing load characteristics.

a) Synchronization Aspect: MyServer can be started as a single threaded or a multi-threaded server. But this decision has to be made statically at the start-up time. Regardless of the selected concurrency policy, the code for synchronization is always present in the executable. The code is responsible for synchronizing the access to the shared list of connections, maintained by the server in order to avoid inconsistency. The "*cserver*" module in MyServer is responsible for the creation and maintenance of a list of connections, which must be handled. The connections are passed on to the threads, which are the objects of the class "*clientsThread*".

```
pointcut virtual dynamicJPS() = execution(
   "int ClientsThread::controlConnections()" ||
   "int ControlProtocol::SHOWCONNECTIONS(...)" ||
   "int ControlProtocol::KILLCONNECTION(...)" ||
   "ConnectionPtr  Server::addConnectionToList(...)" ||
   "int Server::deleteConnection(...)" ||
   "void Server::clearAllConnections()" ||
   "ConnectionPtr Server::findConnectionBySocket(Socket)" ||
   "ConnectionPtr Server::findConnectionByID(u_long)" ||
   "void Server::increaseListeningThreadCount()" ||
   "void Server::decreaseListeningThreadCount()");
 advice synchJPS():before(){
   lserver->connections_mutex_lock();
 }
 advice synchJPS():after(){
   lserver->connections_mutex_unloack();
 }
```

Fig. 5. An aspect for encapsulating the synchronization code

Clearly, in the case, when MyServer is statically set to run single-threaded, the presence of synchronization primitives is unnecessary. To measure the degradation in response time due to the presence of unnecessary locking primitives, the execution time of methods, *"addConnectionToList"* and *"deleteConnection"*, which in total contained two locking and two unlocking primitives, was measured. The server was exposed to a traffic of 1000 requests generated by the Httperf benchmark. The measurement showed that in the case of synchronization primitives present, the average execution time was 34.25 microseconds. However, when the synchronization primitives were commented out, the average execution time was reduced to 24.489. This measurement demonstrated that even when calculated just for two methods, which involved only four locking primitives, 28.49% of the total time was consumed just in acquiring and releasing the locks.

In the current implementation of MyServer, the code for synchronization is scattered in three project modules: *cserver*, *clientsThread*, and *control_protocol*. By using the aspect as shown in Figure 5, the synchronization code was taken out of the different modules and encapsulated as an aspect. It can be seen from the pointcut description in the aspect code that, in total, 10 join points were defined in order to protect all critical data sections.

The isolation and encapsulation of the synchronization code, as a dynamic aspect, was the first step in the transformation of MyServer from a statically configured to a dynamic server, where the synchronization policies could not only be woven when required, but could also be adapted according to the runtime requirements.

An important decision while implementing the synchronization policy is the selected granularity of the protected data segment, for example locking a statement rather than the entire function block, etc. This decision directly relates to the memory and runtime overhead introduced due to synchronization. A selection of a coarse-grained synchronization policy would mean that a large segment of data is protected. This leads to a low memory and runtime overhead due to the locking primitives but increases lock contention. Lock contention

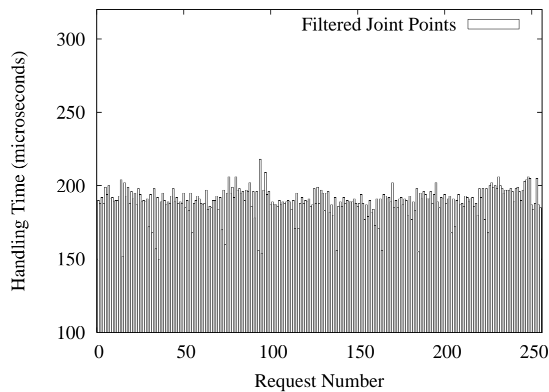

Fig. 6. Response time measurement of MySever with filtered instrumentation

happens when a thread attempts to acquire a lock held by another thread. A coarse-grained synchronization policy is, therefore best suited when the request rate of the clients is low, but leads to severe performance overhead in the case when a large number of concurrent threads try to access the protected data segment. Since the protected data segment is large in the coarse-grained synchronization policy, the offered level of parallelism is low. A fine-grained synchronization policy means that a large number of locks are employed, each controlling access to a very small segment of data. This results in decreased lock contention as it is less likely a thread will request a lock held by another thread. Additionally, a fine-grained policy leads to an increased degree of parallelism, and offers best performance for multiple clients. However, such a policy also means a higher memory and runtime overhead due to many locking primitives, and increased number of executed acquire and release constructs. Additionally, more locks also increase the risk of deadlock.

Clearly, none of the synchronization policies is appropriate under all conditions for a web server. Sometimes, when the traffic is too low, there is no point of imposing overhead due to many locking primitives, and, therefore, a coarse-grained policy would be appropriate. But as soon as the traffic influx increases, a coarse-grained policy would result in bad performance due to decreased degree of offered parallelism. Therefore, as per the changing traffic load, MyServer should be able to switch to the most appropriate synchronization policy. The encapsulation of the synchronization code as an aspect in MyServer allowed to weave and unweave the most suitable policy at runtime. To support the weaving and unweaving at various granularity levels, only the relevant set of join points that correspond to the desired granularity levels were hooked. In order to switch to appropriate synchronization policies, the only thing that needed to be adapted was the pointcut description, which controlled the granularity level. The main implementation of the dynamic aspect, as shown in Figure 5, remained unchanged. Since the coarse-grained synchronization policy affects at the function level, it did not require any re-factoring of the source code of MyServer. The source code was instrumented with *before* and *after* hooks at 10 join points located in different modules that previously contained locking primitives. The version of MyServer, instrumented to support the coarse-grained synchronization policy, was run and applied the same load of 10,000 requests generated by Httperf. It can be seen in Figure 6, that with filtered instrumentation, the server response time was almost the same like that of non-instrumented version. The support for fine-grained synchronization policy proved to be a bit tricky in MyServer, as it required the re-factoring of the source code. Dummy functions had to be introduced in order to be able to apply dynamic aspects at the desired granularity level. The data segments that were required to be synchronized were embedded in the dummy functions. The source code of Myserver was instrumented with a new instrumentation aspect, which introduced hooks only at the dummy functions. The support for fine-grained synchronization policy required the hooking of almost twice the number of join points as compared to the coarse-grained policy. However, still the number was far too less, as compared to full instrumentation approach, and when run and exposed to the same load of 10,000 requests, there was still no identifiable effects on the response time statistics when compared with the non-instrumented version.

The encapsulation of the synchronization code into a dynamic aspect offered a range of benefits to Myserver. First and foremost, this code was not present whenever the server opted for a single-threaded concurrency policy. This meant that MyServer was no longer unnecessarily exposed to the overhead due to synchronization code. The synchronization code was only woven when the server switched from a single-threaded policy to a multi-threaded policy due to traffic load characteristics. The implementation of different synchronization policies only required the tuning of the lock granularity by the redefinition of the pointcut description, whereas the main implementation of the dynamic aspect remained unchanged. This resulted in an efficient reuse of the synchronization policies. The synchronization policies were adapted, via the weaving and unweaving of dynamic aspects encapsulating the policies, according to the traffic load characteristics resulting in an enhanced MyServer performance.

Figure 7 shows the average response times of various versions of MyServer. It can be seen from the figure that the fully instrumented version, in which both call and execution join points were instrumented, performed the worst. The average response time calculated for the fully instrumented version was 232 microsec. Such a performance overhead is clearly not acceptable in many domains. The version, with only execution join points instrumented, performed much better as compared to the fully instrumented version. However, the average response time was still 202 microsec, which was still significant as compared to the average response time of the non-instrumented version, which was 183 microsec. The version with filtered instrumentation, to support only the coarse and fine-grained synchronization policies, offered the best performance. The measured average response time for the filtered version was almost the same as that of non-

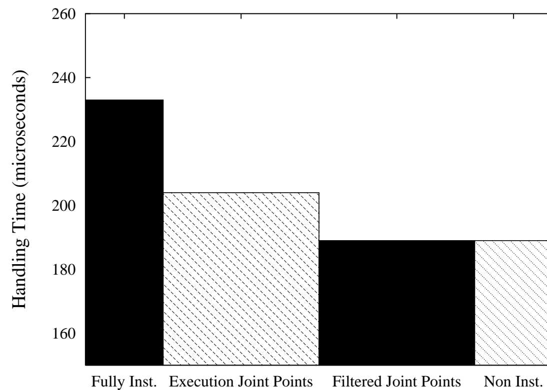

Fig. 7. Difference in response times of various versions of MyServer

```
void * startClientsThread(void* pParam){
   ...
   while(ct->threadIsRunning){
     int ret;
     try{
        Thread::wait(1);
        if((!ct->isStatic()) && ct->isToDestroy() ){
          continue;
        }
        ct->parsing = 1;
        ret = ct->controlConnections();
        ...
     }
     ...
   }
   ...
}
```

Fig. 8. A busy-wait loop implemented in MyServer

instrumented version.

b) Debug Aspect: While running MyServer, it was observed that it was consuming a lot of CPU resources. In order to find out the reason, a dynamic tracing aspect was woven. The tracing aspect revealed that the threads responsible for handling the connections were executing in a busy wait loop. In busy wait technique, a process repeatedly checks to see if a condition is true, such as waiting for keyboard input or waiting for a lock to become available. It is a valid strategy in certain circumstances, such as in the implementation of spinlocks within operating systems designed to run on SMP systems, etc.

Figure 8 shows the method *startClientsThread* in MyServer in which client threads are created. This method contains a while loop that calls the method *controlConnections* which handles the coming connections. After a connection is handled, the control returns to the while loop again. It can be seen from code that even if there is no connection to be handled, the method *controlConnections* is always called in a busy wait fashion resulting in a heavy usage of CPU cycles. Though, there is a wait statement in the code to avoid the full usage of CPU, even this wait statement is problematic as it makes the threads sleep for a fixed amount of time during which they cannot process requests.

```
aspect SynchSem{
      sem_t sp;
      pointcut s_signal() = execution(
       "% Server::addConnectionToList(...)");
      pointcut w_signal() = execution(
       "%ClientsThread::controlConnections(...)");
      SynchSem (){
             sem_init(&sp, 0, 0);
      }
      advice s_signal() : after () {
             sem_post(&sp);
      }
      advice w_signal() : before () {
             sem_wait(&sp);
      }
};
```

Fig. 9. A dynamic aspect for the synchronization of client threads

In order to fix the problem at runtime, a dynamic aspect called *SynchSem* as shown in Figure 9, was woven into the running MyServer. This aspect was meant to replace the busy-wait strategy with a blocking strategy. A semaphor was introduced through this aspect to synchronize the threads. The before advice in the aspect affected the join point *controlConnections*, and performed the wait operation on the introduced semaphor *sp*, which blocked the threads in the case when there was no connection to be handled. By means of an after advice for the join point *addConnectiontoList,* a signal was transmitted via the semaphor *sp* in the case when a connection arrived. This awoke the threads in the block state that then handled the coming connections.

An important point to notice in this case is that the weaving of *SynchSem* aspect did not require additional hooking of new join points, since the set of join points, which were anticipated to be affected by the change of synchronization policies, were already extracted from the source and hooked, as discussed in the foregoing section. The weaving of *SynchSem* aspect at runtime into the server demonstrates the strength and the flexibility offered by the family-based dynamic aspect weaver for runtime debugging and maintenance without causing an additional overhead.

c) Adaptable Concurrency Policies: MyServer starts with a thread-pool policy. The number of threads in the thread-pool are defined statically at the start up time. The static fixing of threads in the thread-pool leads to inefficient and sub-optimal performance of the web server. A very small thread-pool size could result in no benefits at all, whereas a thread-pool with more than required threads introduces substantial overhead due to context switching and also wastes underlying operating system resources. Additionally, if many of the threads in the thread-pool sit idle with no connections to be handled, this constitute to unnecessary overhead. A web site that has a high hit rate but involves lesser processing time, because of light weight requests (static pages or dynamic pages with lower process times) should have a bigger thread-pool as compared to the web sites having low hit rate but heavy weight requests (high processing time). But it is extremely hard to predict the hit rate and the request type at the start up

```
aspect threadPerConn{
  ...
  pointcut thread_Per_Conn() = execution(
    "int Server::addConnection(...)");
  threadPerConn(){
    lserver->nMaxThreads = 30;
  }
  public:
    advice thread_Per_Conn () : before () {
      if(lserver->nThreads < lserver->nMaxThreads) {
         addThread(0);
      }
    }
    ...
};
```

Fig. 10. An aspect to create threads at runtime

time.

None of the concurrency policies is always appropriate under all circumstances. If the request rate is too low, few requests per day, the runtime costs of creating and maintaining a thread-pool may outweigh the benefits of not having to create and destroy threads on the fly (thread-per-connection). As the hit rate increases, switching to thread-pool model could result in better performance and throughput. Therefore, in order to offer an optimal performance, the server should be able to adapt to most appropriate concurrency policy without going offline. The choice of concurrency strategies significantly impacts the performance of web servers subjected to changing load conditions[8]. Such adaptation requirements could be addressed by statically compiling the monitoring code and all policies into the server, and the server could then switch to most appropriate policy at runtime. However, this technique leads to dead code in the running application, and lacks any mechanisms to decouple crosscutting concerns.

To transform MyServer into an adaptable web server, which could switch to a most appropriate concurrency policy at runtime, dynamic aspects *threadpool* and *threadPerConn* were implemented, as shown in Figures 10, and 11. These aspects implement the thread-per-connection and thread-pool policies, and thereby create multiple threads at runtime according to the changing requirements.

In the transformed implementation, MyServer starts with a single-threaded concurrency policy instead of spawning a number of threads in the thread-pool. The code responsible for synchronization is also removed since it is not needed in the case when there is only a single thread. The single-threaded policy remains effective as far as the number of simultaneous connections is low. As the simultaneous request load increases, the server starts showing deterioration in the performance. The decision whether to switch to a thread-per-connection or a thread-pool policy is driven by the coming requests characteristics. If the requests load is not high and the requests are long duration, the aspect *threadPerConn* encapsulating the thread-per-connection policy is woven. But if the number of requests is large and the requests are short duration, it is no longer beneficial to carry on with the thread-per-connection policy due to performance losses caused by the creation and destruction of the threads at runtime. In such load conditions, the policy of MyServer is changed from thread-per-connection to thread-pool by the unweaving of *threadperConn* aspect and the weaving of *threadpool* aspect. As soon as any aspect, which implements a multi-threaded policy, is woven, this changes the concurrency policy of MyServer from single-threaded to multi-threaded. With the transformation to a multi-threaded policy, the aspect that encapsulates the synchronization code as shown in Figure 5, is also woven that is crucial to synchronize the multiple threads being introduced into the system.

The switching to any of the multi-threaded policies also requires to define a value for the maximum number of threads, which can exist in parallel to handle the incoming requests. In original MyServer implementation, the number of maximum threads is defined statically at start up time. But, this value is directly related to the current load characteristics. If the majority of requests are for static pages, this should be set to a higher value and vice-versa. The static setting of this value leads to either under utilization or over utilization of the server.

In the transformed version of MyServer, each of the aspects, which implements any of the multi-threaded policies, also defines the maximum number of threads that would be supported in parallel. This means that dynamic aspects are implemented while taking into consideration the current load conditions and the type of requests. For a thread-per-connection policy, when a request arrives in the system a new thread is created in the before advice of the aspect *thread-pool* if the current number of threads is less than the value of the maximum number of threads introduced by the *threadPerConn* aspect itself. However, in the case of *thread-pool* aspect, the before advice encapsulates the necessary code to create a pool with a required number of threads as shown in Figure 11. It can be seen in the code, that a check is made in the before advice to see if the current number of threads in the server are less than the maximum number of threads the server can afford, and there is no free thread to serve the coming request. The *thread-Pool* aspect provides a dynamic pool implementation, where each time, when there are no free threads available to handle the incoming requests, a specified number of threads are added in the thread-pool. The thread-pool can grow till the value defined by the maximum number of threads (*lserver->nMaxThreads*) the server can afford. Similarly, if the request load decreases, the thread-pool size is decreased accordingly at runtime by weaving dynamic aspects that destroy threads in the thread-pool if they sit idle for some specific time period, thereby freeing up the memory. By the weaving and unweaving of aspects the number of threads in the thread-pool are always tuned according to the current load on the server. The value of maximum threads (*lserver->nMaxThreads*) once defined through the weaving of dynamic aspects, which implement the concurrency policies, can always be tuned by weaving another dynamic aspect that defines this value according to the current load conditions and available resources. The same is true for another configuration parameter, time-out, which is

```
aspect threadPool{
  ...
  pointcut poolcontrol() = execution(
    "int Server::addConnection(...)");
  threadPool(){
     lserver->nMaxThreads = 30;
     ...
  }
  public:
    advice poolcontrol () : before () {
      if((lserver->nThreads < lserver->nMaxThreads) &&
         lserver->countAvailableThreads() == 0)) {
         for(int i = 0;i< poolSize; i++){
            addThread(0);
         }
      }
    }
    ...

};
```

Fig. 11. An aspect to control the size of thread-pool

also adjusted by weaving dynamic aspect, which introduces a value that corresponds to current load conditions and request characteristics.

V. Related Work

There are some web servers and middleware that support switching of concurrency models at runtime according to the load characteristics and system resources. omniORB [1] allows a server to start in a thread-per-connection mode and switches to thread-pool model in the case of higher connection rate to give a very optimized performance. But the limits are set statically at the configuration stage, and are dependent on the number of connections while completely ignoring the characteristics of the connections. dynamicTAO [9] employs a similar approach of employing hooks for the loading and unloading of different concurrency strategies at runtime. JAWS [7] is a dynamically adaptable web server which employs patterns for runtime reconfiguration. It is designed to allow for the runtime customization of the concurrency and event dispatching strategies according to the environmental conditions such as traffic patterns and workload characteristics. OpenWebServer [14] is another web server based on reflection and design patterns, which adapts its concurrency strategies at runtime.

VI. Concluding remarks

Regardless of how well a web server is designed and implemented, the extremely unpredictable nature of the Internet makes it impossible to anticipate and thereby equip the server with the most appropriate policies and strategies. Dynamic aspect weaving offers effective mechanisms for the dynamic adaptation of crosscutting policies. However, none of the available weavers offers a customized support. This limitation coupled with the general absence of a filtration mechanism unnecessarily raises the costs associated with the dynamic aspect weaving mechanism.

The transformation process of MyServer from the static to a dynamically adaptable server started with the integration of MyServer with our dynamic weaving infrastructure, and the aspectization of various crosscutting policies as aspects. We observed that the aspectization process apart from raising the modularization levels of MyServer helped to avoid the runtime overhead due to the execution of synchronization primitives that were executed even if the server was running single-threaded. Our experimentation with MyServer further showed that a large number of join points in the server didn't carry any relevance regarding the adaptation of policies or strategies, and their hooking served no purpose other than causing performance degradation and raising the costs associated with the dynamic aspect weaving mechanism. With the insertion of only 11 hooks, MyServer was transformed into a dynamically adaptable server, which was able to reconfigure its policies, i.e. synchronization and concurrency, and its configuration parameters, i.e. maximum number of threads and time-out, according to the current load conditions and request characteristics. Additionally, this much hooking later proved to be enough to even trace a bug, by weaving a tracing aspect in the running server, and thereby fixing it by weaving another aspect, which otherwise could had required taking the server offline. Our experience with MyServer demonstrated how a powerful join point filtration support in the dynamic aspect weaving infrastructure family helped to transform a statically configured server into one that could adapt and evolve at runtime without imposing any significant memory and runtime overhead.

References

[1] Free High Performance ORB. http://omniorb.sourceforge.net/.

[2] Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[3] C. Bockisch, M. Haupt, M. Mezini, K. Ostermannd, and G. Kiczales. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 83–92, March 2004.

[4] R. Douence, T. Fritz, N. Loriant, J. M. Menaud, M. S. Devillechaise, and M. Suedholt. An expressive aspect language for system applications with Arachne. In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 27–38, Chicago, Illinois, March 2005.

[5] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 51–62, Chicago, Illinois, March 2005.

[6] Wasif Gilani and Olaf Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In *NetObjectDays (NODe '05)*, pages 94–109, Erfurt, Germany, September 2005.

[7] J. Hu and D. Schmidt. *JAWS: A Framework for High performance Web Servers*. John-Wiley, 1999.

[8] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high performance web servers over high speed atm networks. In *Proceedings of the 17th IEEE Conference on Computer Communications (IEEE infocom '98)*, pages 1222–1231, San Francisco, USA, April 1998.

[9] Fabio Kon and Roy H. Campbell. Supporting automatic configuration of component-based distributed systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies Systems (COOTS '99)*, pages 175–188, San Diego, California, 1999.

[10] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. In *1st Workshop on Internet Server Performance (WISP '98)*, 1998.

[11] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just in Time Aspects: efficient dynamic weaving for java. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 100–109, Boston, MA, USA, March 2003.

[12] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for Aspect-Oriented Programming. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 141–147, April 2002.

[13] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, February 2002.

[14] J. Suzuki and Y. Yamamoto. Dynamic adaptation in the web server design space using openwebserver. In *2nd JSSST International Symposium on Object Technologies for Advanced Software '99*, march 1999.

[15] C. Zhang and H. A. Jacobson. TinyC: Towards building a dynamic weaving aspect language for C. In *Proceedings of the 2003 Foundations of Aspect-Oriented Languages Workshop (AOSD-FOAL '03)*, March 2003.