Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems

Sufyan Almajali
Illinois Institute Of Technology
3300 S. Federal St.
Chicago, IL 60616, USA
almasuf@iit.edu

Tzilla Elrad
Illinois Institute Of Technology
3300 S. Federal St.
Chicago, IL 60616, USA
elrad@iit.edu

ABSTRACT
Performance and availability are two critical requirements of today's systems. Current dynamic AOP approaches have addressed the performance issue from one specific dimension: the performance of code after the weaving process. Other performance factors may have a great impact on overall system performance. This includes performance of the weaving process itself and also system availability in single and multithread environment. In this paper, we present Dynamic Aspect C++ (DAC++), a new runtime aspect weaving system that addresses the performance and availability issues in an integrated approach. By addressing the following issues we increase system availability for runtime aspect weaving: We propose a finer-grained of atomic weaving, we introduce two new efficient aspect-weaving techniques and we support multithreading. The scope of performance impact has been extended to include a more general evaluation of the overall system performance. This includes the performance of both the woven product and the weaving/unweaving time. The synergy of these two contributions, increasing availability and overall performance consideration, led to runtime weaving systems of better throughput, response time, and accessibility to resources.

General Terms
Languages, Performance, Design

Keywords
Aspect Oriented Programming, Dynamic Weaving, Language Design.

1. INTRODUCTION
Many dynamic aspect oriented systems have been implemented. Examples of these are PROSE [13], Steamloom [3], Jboss [9], AspectWerkz [17], JAsCo [16], JAC [8], AspectS[7] and others. Some of the characteristics addressed by these systems include the technique used to introduce runtime weaving capability, the types of advices that can be used, the scope of the weaving process like per class or per instance, the ability to, partially or completely define a new aspect at runtime, and the ability to weave and unweave aspect at runtime.

None of the current AOP approaches have addressed the performance impact of runtime weaving at all different times of the program lifetime. Many factors contribute to the overall performance of runtime weaving system. The performance of systems after the weaving process is an important factor because it's the most significant part of the overall system execution time, but other factors may degrade the overall system performance. One of these factors is the runtime weaving/unweaving process. This weaving /unweaving time can lead to performance degradations and poor system availability. Another factor is the weaving atomicity principles followed by these approaches. Not optimizing the weaving atomicity leads to extra degradation in system performance and availability. In addition, the weaving atomicity techniques followed by current systems make it difficult to support dynamic weaving for multithreaded and database transactional systems. A detailed analysis about the atomicity handling by current AOP systems is presented in the coming section.

In this paper, we show that addressing these two issues: (1) performance impact in all dimensions, and (2) weaving atomicity, can lead to a runtime weaving system that exhibits better performance and overall availability.

2. DAC++ Design Goals and Challenges
2.1 Design Goals
The main design goal for the DAC++ system is to support runtime aspect weaving system with the following characteristics: high system performance, high system availability and support of different service levels for the weaving process.

2.1.1 High System Performance
To come up with a high system performance, we need our system to provide efficient execution in the following cases: First, efficiency when the system does not require any aspects – this requires minimizing the overhead of introducing dynamic aspect capabilities so that when aspects are not raised, the execution time is as close as possible to the time of executing the code in a non AOP environment. Second, efficiency after we weave aspects at runtime - this requires runtime to be as close as possible to the runtime at a static woven environment. Third, we require an efficient weaving process and fourth an efficient unweaving process. Weaving atomicity will be discussed latter.

Current AOP approaches provide efficient execution for some of these four cases but not all. Our goal to make our system supports high performance in all of these four cases.

2.1.2 High System Availability

There are three different ways to view system availability: Throughput, Response Time and Access to Resources.

For throughput, the more jobs and transactions the system executes per interval of time, the higher the system throughput. For response time, the shorter time to respond to user requests, the faster the average response time. And finally, the smaller the service interruption time, the better the access to system resources.

Multithreading is a powerful tool for creating high performance applications. Threading is a key feature to maintain high system availability.

Supporting multithreading with the runtime weaving capability can lead to high performance applications that have the flexibility to change at runtime and adapt to different situations.

Some runtime weaving systems support weaving per thread capabilities. Examples are JBoss, AspectWerkz and Steamloom. To our knowledge, the majority of current dynamic AOP systems support coarse-grained atomic weaving. When a weaving request comes to the system, it makes sure that the entire aspect is woven for all targeted joinpoints before proceeding. For single thread applications, that would be easily applicable and there will be little or no effect on the system availability. For multithreaded applications, coarse-grained atomic weaving can cause perceptible degradation to system performance and availability especially if the weaving request is suppose to crosscut multiple threads. In case we have multiple threads running and one or more thread needs a longer time to finish execution, weaving an aspect at application, class or instance level for multiple threads cannot be done before all current running threads finish execution, and no new threads can start before the weaving process is complete. This would be the case when threads share some class and method invocations. Threads may need a long time to finish even if we have low CPU utilization because threads may contain asynchronous operation like I/O operations. Such cases can result in a system with smaller number of jobs served, longer response time and halting services for some time. This availability reduction time will last for the whole period of the weaving time. In addition, it will be always difficult to find the right time to do the weaving step. This could be a very complicated task.

In our system, we have relaxed the condition of atomicity from Coarse-grained atomic weaving to fine-grained atomic weaving. System consistency is still maintained under fine-grained atomic weaving. Using fine-grained atomic weaving simplifies the process of supporting multithreaded systems and minimizes availability degradation during the weaving process. Using fine-grained atomic weaving allows us to weave aspects at any time and at any level (Application, class, instance or thread) without the need to stop the running threads or wait until they finish their execution. Note that we still provide for system consistency (see section 3).

2.2 Design Challenges

To achieve our goals, we had to address the following design challenges.

2.2.1 Fine-Grained Atomic Aspect Weaving

First, let us show by example the difference between fine-grained and coarse-grained atomicity for the runtime weaving system.

Consider the example shown in Figure 1. Figure 1.a shows a program that has two transactions (T1 and T2) and each one of them invokes a number of methods. Assume that while the program is being executed in the middle of transaction T1, a request has been issued to weave aspect A. Assume that weaving aspect A affects methods m1 through m4 and the resulting methods after the weaving are m1A through m4A. If we are using Coarse-grained weaving, then system has to look like Figure 1.b. To keep the atomicity of both transactions, system has to wait until both transactions are finished and then weave aspect A to end up with program in Figure 1.b.

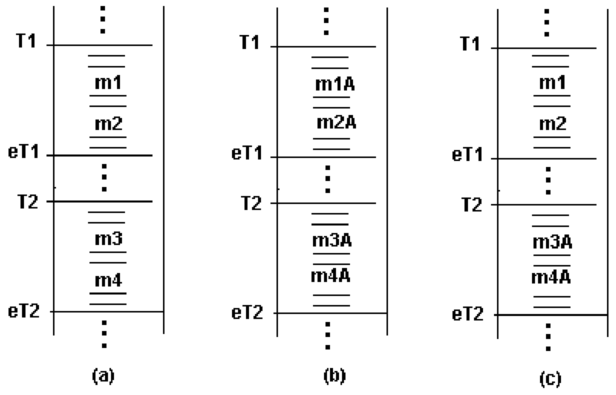


Figure 1. Coarse-Grained and Fine-Grained Atomic Weaving

With fine-grained atomicity, it would be possible to get a case like Figure 1.c where aspect A is woven for T2 but not T1 for a short period of time. This can improve weaving responsiveness and still maintain system consistency. The same thing applies for the multithreaded system in Figure 2. Figure 2.a shows a multithreaded program. Figure 2.b shows coarse-grained weaving and Figure 2.c shows a possible state that may occur sometimes during fine-grained weaving. Figure 2.d shows an invalid system state that should be avoided for single thread and multithreaded systems. Transaction T1 has aspect A partially woven. This state should be infeasible because it does not belong to either the coarse-grained weaving space or fine-grained weaving space.

Figure 3 shows a more complicated atomicity case. This case shows that atomicity needs to be verified at the method level and not only for transactions. Method m1 has in its body two invocations one for m2 and another for m3 (Figure 3.a). If we weave an aspect that targets methods m2 and m3, then the atomic weaving is shown in Figure 3.b. Figure 3.c shows an invalid case that should be avoided.

Achieving fine-grained atomicity is difficult. We need a system that is able to analyze the impact of each aspect on every method and then make this information available at runtime for weaving when needed. Also aspects interaction adds extra complexity to this needed analysis.

2.2.2 Sustaining Both Efficiency and Availability

To produce high system availability, we need a fast weaving approach. Also, we need to support a multithreading-aware runtime weaving system. On the other hand, to produce high performance execution time, we need a weaving approach that results in efficient woven code. Unfortunately, different weaving approaches are available and none of them can guarantee for us both fast weaving process and efficient woven code for all possible system cases.

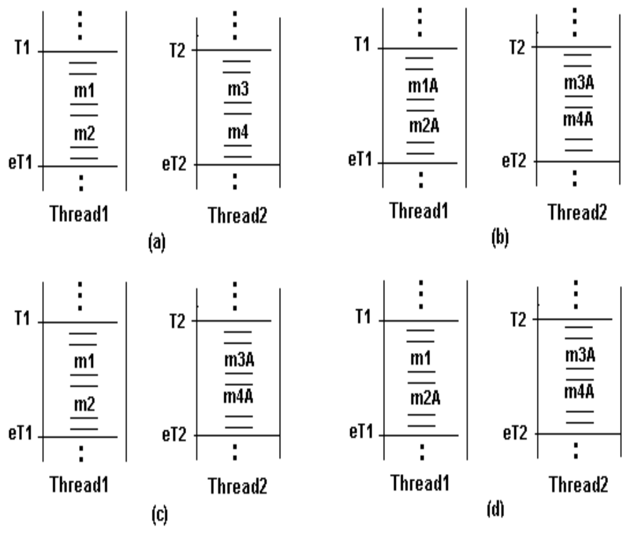


Figure 2. Coarse-Grained and Fine-Grained Atomic Weaving for Multithreading Systems

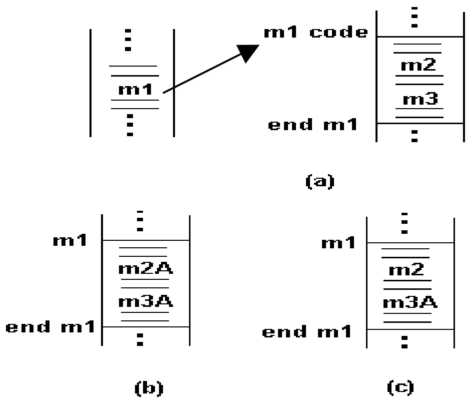


Figure 3. Atomicity at the method level

2.3 Paper Structure

The paper is structured as follows: In Section 3 we describe DAC++ architecture and the different components used and how they are integrated. In addition, Section 3 explains the implementation details of our system. The system performance is examined in Section 4. We conclude the paper with section 5.

3. DAC++ Architecture and Implementation Details

The long-term goal of DAC++ [1,2,4] compiler research is to support a complete feature set of AOP capabilities for networking services, protocols, devices and frameworks.

DAC++ compiler has been implemented to support efficient runtime weaving capabilities for the C++ language. The current version of DAC++ supports a subset of ANSI C++ syntax and extends the C++ language to support AOP capabilities. A customized version of the LCC project [6] has been used as the back end of DAC++ to give us more control over the code generation stage. Part of the back end has been modified to accommodate the needs of our DAC++ compiler.

DAC++ includes five main added elements: Atomicity Analysis for Runtime Weaving, Metadata, Two-Mode Runtime Weaving, Multithreading Support, and the Automatic Availability Analyzer.

3.1 Atomicity Analysis for Runtime Weaving

DAC++ performs Atomicity Analysis for Runtime Weaving at compilation time. Analysis determines the effect of weaving aspects on an application program. Atomicity analysis finds all weaving possibilities that can cause conflict with atomic execution. Analysis results in locating fine-grained atomic execution units. Those units could be methods or transactions. Guaranteeing atomic execution for these units eliminates the atomicity violations that can happen at runtime. Some aspect may cause atomicity conflict and others may not. We consider the aspect as a conflicting aspect only if it can cause side effect problems for method or transaction execution during weaving time. Logging, debugging and aspects of similar functionality are not going to be conflicting aspects in general.

To perform atomicity analysis, the DAC++ compiler performs four steps: control-flow analysis, inter-method data-flow analysis, method execution data-flow analysis and aspect weaving conflict analysis.

3.1.1 Control-Flow Analysis

Figure 4 shows how the analysis happens. In the first step, control flow analysis is used to compute the call graph of the program. Figure 4.b shows the call graph of the code given in Figure 4.a. Constructing call graph is a straightforward process for simple programming languages. In our case, there are three issues that make this process difficult, namely, separate compilation, function overload support and virtual function support.

Separate compilation can be bypassed as an issue by doing call graph construction only when an entire program is presented to a compiler at once, or it can be handled by saving, during each compilation, a representation of the part of the call graph seen during that compilation and building the graph incrementally. We have assumed the first solution although the second can be used in our system. This separate compilation problem applies in the same manner to the rest of steps.

The C++ support for function overloading allows the use of the same name to define different functions of different prototypes.

This problem is resolved using name mangling process. Finally, the problem of virtual function support is the most challenging one. This makes it impossible to create a call graph at compilation time. The reason for that is the need for runtime information to know which virtual function to call for different function call statements. As a result, we had to create an approximated call graph. For each virtual function call, we have created one arrow for each possible targeted virtual function.

3.1.2 Data-Flow Analysis

Next, the compiler performs data-flow analysis. This allows us to gain detailed information about all variables used by every method. C++ supports different type of variables. This includes global, member, static, local and global variables. In addition, our system allows the definition of aspect local variables and advice local variable. Our current system allows aspect to access program global variables, class static variables, and parameters of method invocation, aspect and advice local variables.

The goal of this analysis is to determine for each function call, a safe approximation of the side effects of each method invocation. We represented side effects using two functions: MOD and REF. We computed this for two types of variables supported by C++ global and static.

$\text{MOD}(f,i)$ defines the variable that may be modified by executing the ith instruction of function f.

$\text{REF}(f,i)$ defines the set of variables that may be referenced by executing the ith instruction of function f.

3.1.3 Method Execution Data-Flow Analysis

This step includes accumulating all variables that are used within the method body or its execution flow. The approximated call graph is used in this step along with information generated from previous stage. This allows us to approximate both the MOD and REF functions for each program function.

$\text{MOD}(f)$ defines the set of variables that may be modified by executing function f. Also, the MOD function defines for each variable the number of modifications. $\text{MOD}(f)$ takes the following form:

$$\text{MOD}(f) = \{(\alpha 1, \sigma 1, \xi 1)\ ,\ (\alpha 2, \sigma 2, \xi 2)\ ,\ ......\ ,\ (\alpha n, \sigma n, \xi n)\}$$

where αi is the name of ith variable in the list, σi is the type of the variable (global or static), and ξi is the number of times variable αi can be modified inside this method or method execution flow. For each variable i, its name αi is fully qualified using the function or class mangled name.

Regarding the REF function, $\text{REF}(f)$ defines the set of variables that may be referenced by executing function f and the number of references for each variable. $\text{REF}(f)$ can be represented by the following form:

$$\text{REF}(f) = \{(\alpha 1, \beta 1, \zeta 1)\ ,\ (\alpha 2, \beta 2, \zeta 2)\ ,\ ......\ ,\ (\alpha n, \beta n, \zeta n)\}$$

where αi is the name of ith variable in the list, βi is the type of the variable (global or static), and ζi is the number of times variable αi may be referenced inside this method or method execution flow.

MOD and REF functions are easy to compute for functions that do not contain other function calls. For nested function calls, the approximated call graph is used. Also, MOD and REF functions will be evaluated for the advice function for each aspect.

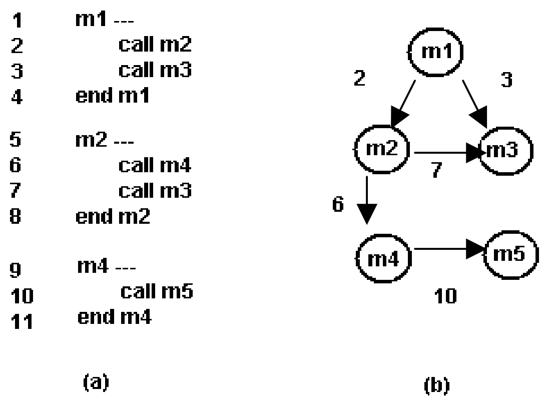
3.1.4 Aspect Weaving Conflict Analysis

The purpose of the final step is to study the side effect impact of each aspect in every method. Assume we want to weave aspect A with method M, then we have two cases:

Case 1: Aspect A does not cause side effects to the execution of the application.

Case 2: Aspect A does cause side effects to the execution of the application.

In case 1, we do not need to force atomic weaving because it is not required. In case 2, we need to force atomic weaving to avoid partial weaving of aspect A. To satisfy atomic weaving during the weaving of aspect A, the system should be in one of two states: 1) either run the effected application part without any aspect A side effects or 2) run it with same aspect A side effects.



```
1     m1 ---
2          call m2
3          call m3
4     end m1

5     m2 ---
6          call m4
7          call m3
8     end m2

9     m4 ---
10         call m5
11    end m4
```

Figure 4. Control-flow Analysis: Call-graph

The following algorithm is used to determine the aspects require atomic weaving and the program location for that.

Assume we have aspect “A” that crosscuts application methods represented by the set $S_A = \{M_1, M_2, M_3, .. M_n\}$.

(1) Assign Aspect “A” a unique version number V_A.

(2) For each method $M_i \in S_A$, use the approximated call-graph and find all execution paths that lead to M_i. Execution paths can be cached for later use and avoid redundant computation. The result of the second step will be a set of execution paths, each one of the following form: $(K_1 K_2 .... K_h M_i)$.

(3) Evaluate the following step for each execution path $(K_1 K_2 .... K_h M_i)$:

```
conflict = false;   count =1;
    while (count <=h)
{
$S_\alpha = REF(K_{count}) \cap MOD(A)$
(i) if $S_\alpha = \varnothing$ then
exit; // break while loop , no conflict

(ii) if ($\sum \forall \beta_i$ for $\alpha_i \in S_\alpha$
= $\sum \forall \beta_i$ for $\alpha_i$ for all $K_{count}$ instructions before invoking $K_{count+1}$)
exit; // break while loop , no conflict  ($S_\alpha \neq \varnothing$ implicitly)

(iii) if ($\sum \forall \beta_i$ for $\alpha_i \in S_\alpha$
= $\sum \forall \beta_i$ for $\alpha_i$ for all $K_{count}$ instructions after invoking $K_{count+1}$ )
exit; // break while loop , no conflict
(iv) if ($\sum \forall \beta_i$ for $\alpha_i \in S_\alpha$ = $\sum \forall \beta_i$ for $\alpha_i$ for REF($K_{count+1}$) ) then
     count=count+1
     else
    {  conflict=  true;
        record Aspect version number "$V_A$" as a conflict with current execution path. And force atomicity at method $K_{count}$
          exit; //  break while loop , conflict case}
  } // end of while statement
```

S_α defined in step 3 represents the set of variables used by method K_i and may get modified by aspect A. Condition (ii) of step 3 addresses the case when all of K_i instructions that reference the variable(s) modified by the aspect are before invoking method K_{i+1}. This leads to no conflict since the aspect A will be woven after all references happen. Condition (iii) addresses the opposite situation. Conflict will happen when some references are made before invoking method K_{i+1} and the others are made after invoking method K_{i+1}.

One important issue here is aspect interaction. When we weave an aspect A and follow it by weaving an aspect B, this needs more complicated analysis. It is possible that weaving aspect A or B alone does not cause conflict with method M, but weaving both aspects causes conflict with the method.

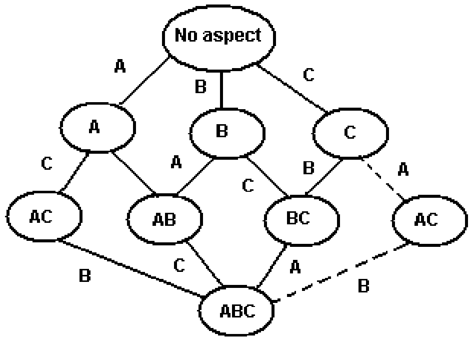


Figure 5. Aspect Interaction Analysis for Aspect A, B and C

To resolve this issue, DAC++ generates approximated call-graphs for all weaving possibilities of the aspect that have side effects (write statement(s)). Figure 5 shows aspect interaction for three aspects A, B and C. Each node represents a weaving case. Each node is given a version number to represent aspects being woven for the case.

For each node in the graph, DAC++ performs conflict analysis and records the conflicting version number with each method. Notice that this could be a space and time consuming process in case we have a large number of aspects, but it is needed only at compilation time (done one time) and the result graphs are not needed at runtime.

3.2 DAC++ Metadata

DAC++ metadata includes all information needed to support atomic runtime weaving/unweaving. It is the aspect runtime representation. DAC++ generates the initial metadata during program compilation and loading and linking time. Metadata information changes during program runtime to reflect the aspect weaving and unweaving status. Metadata includes three main categories: classes and methods metadata, aspect metadata and threading metadata.

Part of the metadata is method relocation entries. They are compiler-generated entries and used to track all locations that access the method. Many compilers generate relocation entries information [11]. This information is usually used by linkers to link multiple program modules. They are usually used at linking time, but we filter the unneeded relocation entries and load the rest in our system and utilize them at runtime.

3.3 Two-Mode Runtime Weaving

One of the main unique things used to support efficiency in our system is the inclusion of two approaches for runtime weaving. The two weaving approaches represent two different system modes for weaving aspects at runtime. DAC++ uses these two modes alternatively to meet the different system needs at different times for different weaving cases. Weaving modes include the following: Wrapping weaving mode and Splicing weaving mode.

Both modes share the first steps of the weaving process and differ in the rest of the steps. One exceptional characteristic of DAC++ weaving modes is their minimum service suspension time and the promptness of the weaving process without affecting program semantics or performance.

3.3.1 Wrapping Weaving Approach

Figure 6 shows how the wrapping mode works. When the AOP thread receives a weaving request, aspect metadata will be used to determine the targeted methods of this weaving process. For each one of these methods, DAC++ instantiates and customizes mini stubs of code that will be used for temporary fine-grained atomicity check and method call redirection. In addition, DAC++ creates a wrapper method that does not have any code but method invocations and two stack instructions. The wrapper method structure is formulated to reflect the type of advice: before or after. This whole process happens without the need to stop any program part to continue its execution in parallel. Also, DAC++ can control the percentage of CPU cycles given to the AOP weaving thread so we have minimum impact on system availability. After having the stub and the wrapper methods created simultaneously with program execution, the affected thread(s) will be suspended for very short period of time to do runtime relocation for the methods. This last step is actually runtime relinking and it is the only stage where we have to suspend the execution. Suspension can be done one thread at a time to have minimum impact on overall system availability. After this minimal service suspension time, thread(s) resume their execution from the state they were suspended at without losing any state information.

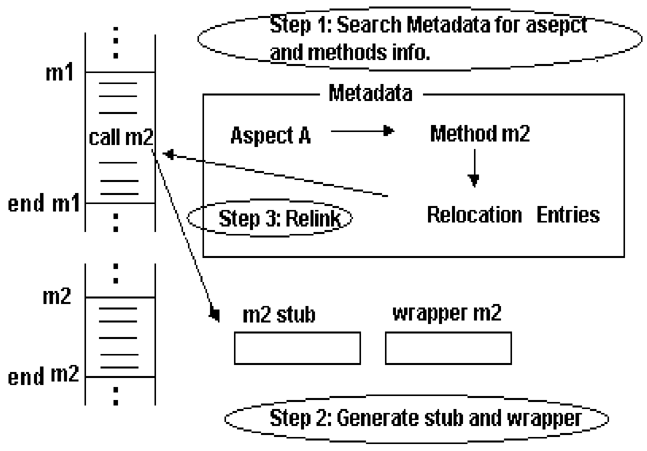


Figure 6. Wrapper Weaving Approach

The runtime relinking step allows method joinpoints to point to the stub method or wrapper method depending on atomicity conflict status. The stub takes care of atomicity checking. For each aspect weaving step, there is one Boolean variable safe_to_weave (default is false). Also, there are three types of stubs: first-step-weaving stub, atomicity-check-and-relink stub and atomicity-disable stub.

In case of weaving a non-conflicting aspect (an aspect that does not cause conflict in any case), the relocation entries are mapped to point to the wrapper method directly and stubs are not be used at all.

In case of weaving a conflicting aspect (aspect may cause atomicity conflict), the first-step-weaving stub, the atomicity-check-and-relink stub and the atomicity-disable stub are used as follows: The relocation entries are mapped to point to the first-step-weaving stub directly.

The first-step-weaving stub does the following tasks:

1) Check the stack to evaluate the current calling sequence and discover which methods are activated (discover current execution path).
2) If current aspect weaving version V_A conflicts with the current execution sequence, then set safe_to_weave to false otherwise, set safe_to_weave to true. Method causes conflict is predetermined at compilation time.
3) For that method stack frame, save its return address, and modify it to point at the atomicity-disable stub.
4) If safe_ to_weave then
   a. Relink caller to new wrapped_method
   b. Execute_wrapped_method
5) Else
   a. Execute_original_method
6) Replace first-step-weaving stub by atomicity-check-and-relink stub

Note that the first-step-weaving stub is executed one time only for each weaving request.

The atomicity-check-and-relink stub has only steps 4 and 5 of the first-step-weaving stub.

The atomicity-disable stub will be called only when system reaches the end of the method that has a conflict. The atomicity-disable stub will check again for atomicity conflict with the current stack and either reset the return address of the conflicting method as happened in the first step or, set the safe_to_weave variable to true. In case safe_to_weave is set to true, the next method calls will access the wrapper method directly and the stubs will not be accessed.

The relink step used by the DAC++ stubs allows next method calls to be direct method calls instead of redirecting each call through the stub. Also, to accomplish relinking, we do not need to have any information saved, the caller calling instruction address can be easily computed from the current stack frame of the current executing method.

The wrapper method contains the following assembly code:

1 **Wrapper_method**:

2 **Call** advice_function ; used if the advice is of type *before*

3 **Pop** some_memory_loc; this is used to pop the return

4 ; address of the wrapper method

5 **Call** original method ;

```
6   push some_memory_loc; replaceing the return address into its
7                                  ; correct location
8    Call advice_function ;  used if the advice is of type after
9   Leave
10  Ret
```

Either line 2 or 8 will be used but not both depending on the advice type. The reason for using the push and pop operations is to avoid pushing the method parameters again.

The wrapping weaving approach has the advantage of constant weaving time per method. The overall time complexity will be linear in the number of targeted methods affected by the weaving process. On the other hand, this approach adds little overhead during method execution because the wrapper method adds one indirect call step in addition to the two additional instructions, pop and push.

3.3.2 Splicing Weaving Approach

To reduce overhead completely, we have included another weaving approach that avoids indirect methods calls. We call this approach: splicing weaving approach. In this method, we follow the same start in the same manner as with the wrapping weaving approach by including a stub to check for atomicity conflicts. But, instead of using a wrapper method, DAC++ generates a new version of the method that contains the method code after the weaving process. Splicing the method code with the advice code on the fly does the process. Methods splicing does not need recompilation or method/class reloading. The original method code is already in memory. DAC++ reuses a copy of the method code and makes a small modification to accomplish the splicing with the aspect advice code. Splicing is done as the following:

1) Initially, the method object for X86 machines has the following format:

```
method_label:
      enter  X,0
  ; method code
       leave
       ret
```

Enter instruction is used to perform the method prologue by initializing needed registers and reserving Local_Bytes in the stack frame for local variables. Leave is used to perform the program epilogue.

2) The advice code has similar code:

```
advice_label:
      enter  Y,0
  ; advice code
       leave
       ret
```

3) The spliced code looks like the following:

```
splicied_method:
      enter  X+Y,0
  ; relocated advice code if before advice
  ; method code
  ; relocated advice code if after advice

       leave
       ret
```

A challenge issue for machine code splicing is its dependency on the memory location. Copying a method from one memory location to another is simply not going to work. To solve this problem, DAC++ offers two solutions. The first solution is to customize the back end of our compiler to generate method code that is location independent. Position Independent Code (PIC) is a technique used by some operating systems to load code without having to do load-time relocation and to share memory pages of code among processes even though they do not all have the same address space allocated [11]. PIC code does not help our situation because we need to relocate method code only and not the entire program. Consequently, we had to generate a kind of position independent method code (PIMC). Generating PIMC code is much easier as a process than PIC code. In PIMC code, all jump and branch instructions are made relative. But unlike PIC, we do not need to take care of changing instructions that have direct data access.

Although PIMC allows us to implement code splicing easily, PIMC has little overhead due to the relative referencing that it uses. For each jump and branch instruction, an extra memory reference is needed in order to access the PC register that saves the current execution address.

DAC++ offers programmers the option to generate PIMC code with version 2. The PIMC code of version 2 does not have relative address access instructions. Instead, it uses direct address access instructions. To make it a PIMC code, DAC++ attaches a list of memory addresses that need to be shifted during method copying and splicing time. This eliminates overhead completely but requires more space.

The same idea is applied to the advice code. The access to local variables need to be relocated by X amount because the frame pointer will be pointing at the original method local variables.

User can pick the optimization option based on the application needs and code size.

3.4 Multi-Threading Support

DAC++ supports runtime weaving for multi-threaded applications. Support includes POSIX threads, a well-known threading standard in the C++ and C languages. DAC++ allows weaving per thread. During program compilation, DAC++ identifies each thread type and performs the following steps: First, it uses the result of the call-graph generated during atomicity analysis to find all methods used by each thread and thread flow. After this, it compares all applications threads and sees the methods shared among the different threads. For each one of these methods, one copy or more of the method source code is created and method renaming is applied. The original method name is saved in the metadata part and renaming happens only for the mangled name. As a result, each thread will end up with its own code space. This allowed us to implement aspect weaving per thread without paying performance penalty. Figure 7 shows the way in which threads are handled. It is important to notice that the code space conflict has to be resolved for the inner method calls too. In case of a thread calling method m1, and method m2 is called inside m1, then DAC++ treats m2 in the same way it treats m1 with respect to threading handling. This repeats for all user-defined methods but not built-in functions like the mathematical sin function. We did not include support for per thread weaving with built in functions although it can be added to our system. It is interesting to know that cflow and weaving per instance are both supported using the same technique used for threads.

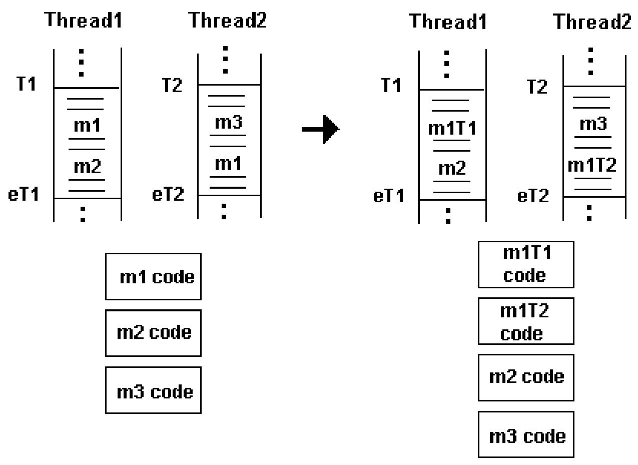


Figure 7. Threading Preprocessing

4. DAC++ Performance

We have measured the performance of DAC++ using different measurements. All evaluations were done using 3.0 GHz Pentium IV on Linux machine with I GB memory. First, we measured the overhead added by DAC++ when no aspect is woven. Table 1 shows the result of this evaluation. PIMC version 2 overhead is very minimum. The little overhead we have is due to the lack of support of all optimization levels by standard C++ compilers as we described in previous section. PIMC version 1 code has more overhead specially when it comes to loops and jumps instructors. This is expected because of the relative referencing used by this version of code generation.

Table 1. DAC++ Overhead When no Aspect is woven

Benchmark	DAC++ PIMC v1	DAC++ PIMC v2
Method call	10.02%	1.7%
Fibonacci	11.02%	2.03%
Nested Loop	15.02%	3.2%

In the second performance measurement, we evaluated the performance overhead of the two weaving approaches supported by DAC++ and compared them to statically woven aspect in AspectC++ system [15]. Table 2 shows the performance overhead after dynamically weaving aspect at run time.

Table 2. DAC++ Overhead of Weaving Aspect at Runtime

Benchmark	WWM / PIMC v1	WWM / PIMC v2	SWM / PIMC v1	SWM / PIMC v2
Logging Aspect	7.1%	2.11%	6.02%	1%
Fibonacci Aspect	9.4%	3.53%	8.02%	2.03%

In general, Splicing Weaving mode (SWM) with PIMC version 2 has the lowest performance impact. Notice that Wrapping Weaving mode (WWM) has more performance overhead and the reason is the indirection of method call and the additional parameter passing and handling necessary to accomplish the wrapping.

5. Conclusion and Future Work

DAC++ supports runtime weaving with new capabilities that are not offered by current approaches. It supports per application, per thread, per class and per instance weaving. In addition, it controls the weaving process so that it does not impact the overall system availability. It offers a minimum service suspension time with very efficient woven code. With its runtime relinking and fine-grained weaving capabilities, DAC++ makes the promptness of the weaving process faster with minimum program suspension.

The current DAC++ implementation has a set of limitations that need to be elaborated in future. First, We are planning to extend DAC++ capabilities in future to support full C++ syntax. Also, redefining the pointcut designator for an aspect at runtime is problematic for DAC++. We prevented it because it needs extra space at runtime and it slows down the weaving process due to the need for atomicity analysis.

Another issue that we need to address is the compiler optimization techniques that fit with DAC++. Aggressive optimization levels supported by current C++ compilers have note been addressed yet. The optimization level of spliced code generated on the fly is not high compared to the code generated during compilation. Run time code optimization is needed to achieve better performance with woven code.

One of the biggest challenges to runtime weaving in C++ is to support inter-type aspect runtime weaving. We needs to modify the C++ object model at runtime. This needs a major change in compiler technique.

Finally the effect of the program execution on aspect semantics using fine-grained atomic weaving needs more elaboration.

6. ACKNOWLEDGMENTS

This work is supported in part by CISE NSF grant No. –0137743.

7. REFERENCES

[1] Almajali, S. and Elrad, T. A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach. *In Dynamic Aspect Workshop*, Lancaster, England. March 2004.

[2] Almajali, S. and Elrad, T. Dynamic Aspect Oriented C++ for Upgrading without Restarting. *In proceeding of Conference on Advances in Internet Technologies and Applications with Special Emphasis on E-Education, E-Enterprise, E-Manufacturing, E-Mobility, and Related Issues*, Purdue, USA, July 8-11, 2004.

[3] Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K. Virtual Machine Support for Dynamic Join Points. In AOSD 2004 Proceeding. ACM Press, 2004

[4] DAC++ Home Page. http://www.iit.edu/~almasuf/dacpp.html

[5] Douence, R. and Sudholt, M. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[6] Fraser, C. and Hanson, D. A Retargetable C Compiler: Design and Implementation. . Addison-Wesley, 1995.

[7] Hirschfeld, R. Aspect-Oriented Programming with AspectS. http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/Docs/AspectS_NODe02_Erfurt2_rev.pdf

[8] JAC Home Page. http://jac.aopsys.com

[9] JBoss AOP Home P. http://www.jboss.org

[10] Levine, J. Linkers and Loaders. Morgan-Kauffman, October 1999.

[11] Lippman, S. Inside the C++ Object Model. Addison-Wesley, 1996.

[12] Popovici, A., Gross, T. and Alonso, G. Dynamic Weaving for Aspect Oriented Programming. In AOSD 2002 Proceedings. ACM press, 2002.

[13] Popovici, A., Gross, T. and Alonso, G. Just-in-Time Aspects. In AOSD 2003 Proceeding. ACM Press, 2003

[14] Schult, W. and Polze, A. Dynamic Aspect Weaving with .NET. http://www.dcl.hpi.uni-potsdam.de/dcl/papers/GI2002.ps

[15] Spinczyk, O. ,Gal, A. and Schröder-Preikschat, W. AspectC++: An aspect-oriented extension to the C++ programming language. *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of Conferences in Research and Practice in Information Technology. ACS, 2002.

[16] Vaderperren, W. and Suvee, D. Optimizing JAsCo dynamic AOP through HotSawp and Jutta. *In Dynamic Aspect Workshop*, Lancaster, England. March 2004.

[17] Vasseur, A. Dynamic AOP and Runtime Weaving for Java-How does AspectWerkz address it*? In Dynamic Aspect Workshop*, Lancaster, England. March 2004.