TinyC^2: Towards building a dynamic weaving aspect language for C [*]

Charles Zhang and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
10 King's College Circle
Toronto, Ontario, Canada
{czhang,jacobsen}@eecg.toronto.edu

ABSTRACT
The runtime behaviors of software systems are often subject to alteration or intervention after their development cycles for various reasons such as performance profiling, debugging, code specialization, and more. There are two separate domains related to the instrumentation of software systems, one being various performance measurement and instrumentation tools, the other the new aspect oriented programming (AOP) paradigm. This paper describes TinyC^2 language, a language approach which experiments with the idea of implementing an aspect oriented language based upon existing system instrumentation techniques. Like other aspect oriented languages, TinyC^2 uses new language constructs to allow programmers to intentionally compose systems in the dimensions of both components and aspects. In this paper, we discuss both the grammatical features and the compiler architecture of the TinyC^2 language. Through the TinyC^2 implementation, we demonstrate that a language approach can well bridge the gap between the AOP paradigm and the existing system instrumentation technologies. It greatly simplifies code instrumentation effort and provides runtime optimization at the application level.

Keywords
Aspect Oriented Programming, Compiler, Dynamic Instrumentation, Dynamic Weaving, Source-to-source translation
Proceedings

1. INTRODUCTION
Programming methodologies have evolved from direct machine-level coding to object-oriented programming. Good modularization capability in the programming language design allows software architects to successfully tackle two issues: the ever growing complexity of software systems and the increasing diversity and volatility of the execution environment. Besides improving language designs, there has also been extensive work on finding better compiling techniques to provide effective adaptations for software systems and to efficiently support a wide spectrum of hardware platform and computing resources [6, 5] that change dynamically. However, compiler-based program adaptation and optimization techniques are powerful but limited if the optimization involves changing the functional behavior of the system. These optimization tasks include adaptations to many domain specific characteristics, such as state validation conditions, synchronization strategies, logging strategies, and many others. It is very difficult to build compilers to make such application level decisions flexibly.

To overcome this difficulty, it becomes necessary to perform post-development transformation to large software systems according to specific usage scenarios. The post-development transformation mainly includes modifications made to software systems after their development cycles. A major stream of manipulation techniques includes tools that provide source-code level instrumentation, as in SvPablo[1], and post-compilation instrumentation techniques as in jContractor[2] and Vulcan [4]. Dyninst [7] and the Paradyn[3] performance tools provide runtime instrumentation to C/C++ systems. Another stream of program manipulation techniques mainly belong to the aspect oriented programming paradigms [9], where "instrumentation" has the first-class status in the language design and can be used to compose system functionality. AOP advocates composing systems using different sets of models and leaving the integration work to the AOP compiler which is also referred to as the aspect weaver.

Code instrumentation techniques and aspect oriented programming are two fields that are developed independently. We think that those two domains are fundamentally com-

[*]In: Foundation of Aspect Oriented Languages Workshop in conjunction with 2nd AOSD Conference 2003, Boston, MA.

[1]SvPablo: A Graphical Source Code Browser for Performance Tuning and Visualization `http://www-pablo.cs.uiuc.edu/Project/SVPablo/SvPabloOverview.htm`
[2]Java Implementation of Design By Contract for the Java Language `http://jcontractor.sourceforge.net/`
[3]Paradyn. `http://www.cs.wisc.edu/~paradyn/`

patible as they both perform a certain type of after-the-fact transformation to the existing software systems. An aspect oriented language provides a more powerful approach in terms of methodology. We think that the various code instrumentation techniques can be treated as means to realizing the methodology in practice. The main motivation of our work is to experiment with such ideas by developing an aspect language using existing code instrumentation techniques. The advantage of using a hybrid language is twofold. Firstly, a hybrid language design which decouples the language semantics from the backend implementation platform can increase the configurability and the adaptability of the aspect language. The compiler is able to readily take advantage of the advances in the code instrumentation domain by selecting different lower-level implementation strategies to instrument the system, i.e. to weave aspects, under different circumstances. Secondly, since a language provides a high level abstraction of the instrumentation semantics, it is easy to understand, to change, and to maintain the instrumentation code. This technique is also applied in [3] and [8].

The second motivation of our work is that for most of the AOP languages today, including AspectJ[4], Hyper/J[5], AspectC[6] and AspectC++[7], the transformation of programs is done statically either at the source-code level or at the byte-code level. To maximize the benefit of multi-dimensional programming, it is desirable to have the support for dynamic transformation since a lot of platform specific parameters are not available until runtime. HandiWrap [1] is a runtime weaving aspect language for Java. In the C/C++ programming domain, we are not aware of any previous work in aspect languages that provide dynamic weaving. The runtime weaving property is directly supported by the Dyninst library. We are interested to see how an aspect oriented language can take advantages of platforms like Dyninst in supporting dynamic adaptations.

We have developed the TinyC^2 language, which is a prototype aspect language. The language is designed to be an extension of the C language with new language constructs to enable the composition of aspect programs. This is also a common language design approach used in AspectJ and AspectC++. The compiler of TinyC^2 is essentially a source-to-source translator that translates C statements to the API instructions of the target instrumentation tool. We construct the compiler to be independent of any particular instrumentation techniques, thus, give the compiler the flexibility of switching to different instrumentation tools. Currently, we have implemented support for the Dyninst runtime instrumentation platform. Due to the runtime instrumentation nature of Dyninst, TinyC^2 can be treated as the runtime weaving aspect language.

The rest of the paper is organized as follows: Section 2 presents the related work regarding aspect oriented language designs. Section 3 presents a detailed description of the new language features of TinyC^2. The architecture of the compiler is also discussed in this section. Section 4 uses three case studies to demonstrate the effectiveness of the dynamic weaving nature of TinyC^2 in addressing runtime crosscutting concerns. Section 5 presents runtime characteristics of TinyC^2. Section 6 concludes the paper.

[4]AspectJ `http://www.aspectj.org`

[5]HyperJ `http://www.alphaworks.ibm.com/tech/hyperj`

[6]AspectC `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`

[7]AspectC++ `http://www.aspectc.org`

2. RELATED WORK

There are a number of aspect oriented programming languages in C and Java flavours. AspectJ adds an aspect oriented extension to the Java programming language. **Aspect**s are AspectJ's units of modularity. They are defined in terms of pointcuts, advice, and introductions. By adding these simple constructs, AspectJ enables the clean modularization of crosscutting concerns such as synchronization, context-sensitive behavior, and multi-object protocols.

Hyper/J is developed by IBM. It also supports multi-dimensional separation of concerns for Java. It provides the ability to identify concerns, specifies modules in terms of those concerns, and synthesizes systems and components by integrating those modules. It operates on standard Java class files, without need of source, and produces new class files to be used for execution.

AspectC++ is an application of the AspectJ approach to C++. It is a set of C++ language extensions to facilitate AOP with C++. It provides language features that allow a highly modular and thus easily configurable implementation of monitoring tasks and supports reuse of common implementations. AspectC++ offers virtual pointcuts and aspect inheritance to support the reuse of aspects. AspectC is an extension to the C language based on the AspectJ technologies. It is being developed concurrently with the a-kernel[8] project at UBC.

MDL [12] is a language built by the authors of Dyninst. It is specifically designed for performing runtime instrumentation using the Paradyn runtime code generation platform. The language is specialized for writing instrumentation requests in terms of performance metrics. The MDL code is parsed and translated to Paradyn instructions. Although the authors of MDL do not mention AOP, since their language can capture crosscutting concerns, we categorize it as one type of aspect language.

3. THE TINYC^2 LANGUAGE

The design goal of the TinyC^2 language is to provide a language perspective in terms of code instrumentation, and, at the same time, to establish a framework for implementing a post-compilation weaving aspect language that uses the C syntax and a third party instrumentation tool as the backend. The rest of the section describes the language in detail from both the syntactic point of view and the compiler architecture perspective.

3.1 Language Features

Using aspect oriented programming terms, the component programs of TinyC^2 can be composed in the C language. The aspect program is composed using TinyC^2. Similar to AspectJ, TinyC^2 implements standard C grammar rules

[8]a-Kernel `http://www.cs.ubc.ca/labs/spl/projects/a-kernel.html`

with the addition of a few new syntactic constructs. Programmer can use the regular C syntax to compose code blocks. However, the basic modularization units in TinyC^2 are not functions as in C but "snippet"s. A snippet is a unit of aspect implementation. It encapsulates a code block and defines the "weaving" points in the component program where the aspect code is inserted. Snippets are functionally equivalent to the "joinpoint" and "advice" concepts in an "aspect" module in AspectJ.

```
1   void trace(char *);
2   onentry Service(int size) : (int totalsize)
3   {
4     trace("function service is called\n");
5     if(size>0)
6     {
7           totalsize=totalsize+size;
8     }
9
10  }
11  onexit int retv Service(int size) : (int totalsize)
12  {
13    trace("function service is exiting\n");
14    if(retv<0)
15    {
16          totalsize=totalsize-size;
17    }
18
19  }
20
```

Figure 1: Snippet: onexit and onentry constructs

Let us look at the constructs of "snippet"s more closely through Figure 1. This code snippet illustrates how to implement the typical logging and tracing functionality as an aspect program in TinyC^2. This aspect program, like in regular C programs, first declares the prototype of the function `trace` (Line 1). The first section of the program (Line 2-10) traces the invocations of the function `Service` in the target system. That is, before the `Service` is executed, a message is logged (Line 4) and the size is added to a total size (Line 7) if the size is bigger than zero. More specifically, the `onentry` construct is defined as follows:

`onentry FunctionName` (*formals_list*) : (*formals_list*)

The construct binds the following identifiers in the component program: 1. function names and these formal parameters (arguments); 2. global variables in the component program designated by the formals after the ":".

The second code segment (Line 11 - 19) presents an example of the construct `onexit`. This snippet logs a message before the function `Service` returns. It also performs some post-invocation checking so that, if the `Service` function returns a negative value possibly meaning an error, the service size is subtracted from the total size. The `onexit` construct can be used to insert new behavior after a certain function finishes executing. We define the syntax of `onexit` as follows:

`onexit` *formal_list* `FunctionName` (*formals_list*) : (*formals_list*)

The difference of the `onexit` construct as comparing to `on entry` is that `onexit` allows us to bind to the return value of the function which is designated by the *formal* before the function name. The formal grammar definition of these two "snippet" constructs are defined using EBNF in Figure 2 and Figure 3.

```
onentry
  :  TK_onentry ID LPAREN
     (formalParameter (COMMA formalParameter)*)?
     RPAREN COLN LPAREN
     (formalParameter (COMMA formalParameter)*)?
     RPAREN
     block
```

Figure 2: Grammar definition for onentry

```
onexit
   :  TK_onexit (formalParameter)? ID
     LPAREN
     (formalParameter (COMMA formalParameter)*)?
     RPAREN COLN LPAREN
     (formalParameter (COMMA formalParameter)*)?
     RPAREN
     block
```

Figure 3: Grammar definition for onexit

Currently, the TinyC^2 provides a simple pattern matching mechanism based on the prefix of the function names and their return types. In addition, the wildcard character "*" can be used to match all functions. The prefix-based matching can be extended to the regular-expression-based matching. The pattern can be defined using the "`group`" keyword as follows (the vertical line denotes an OR relationship):

`onexit | onentry` *formal_list* `group prefix_of_function | *` (*formals_list*) : (*formals_list*)

Currently, TinyC^2 supports integer and character computations. It supports conditional statements such as `if` and `else`. The `for` and `while` loops are also supported by the language.

3.2 Compiler Architecture

Generally speaking, the compiler of TinyC^2 is essentially a source-to-source translator built on top of the ANTLR[9]parser generator tool, formerly known as PTTCS. ANTLR uses a LL(K)-based language parsing scheme to parse a grammar file and generates the corresponding parser. The TinyC^2 compiler consists of three main components: the grammar file for the language, the lexer and parser generated from the grammar file, and the backend code translator and generator. Programs written in TinyC^2 language are translated by TinyC^2 compiler to a source file written according to

[9]ANTLR: ANother Tool for Language Recognition. http://www.antlr.org

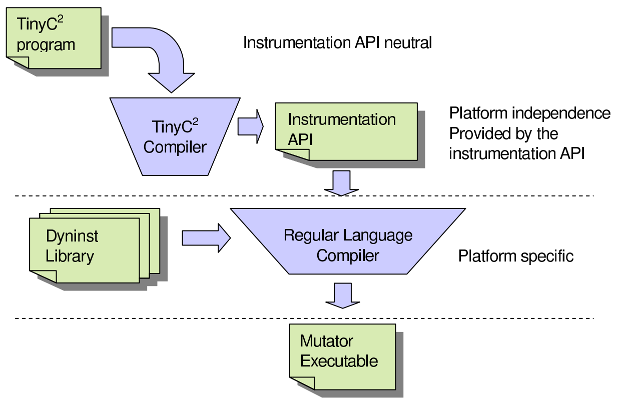
TinyC^2 program
Instrumentation API neutral
TinyC^2 Compiler
Instrumentation API
Platform independence Provided by the instrumentation API
Dyninst Library
Regular Language Compiler
Platform specific
Mutator Executable

Figure 4: Compilation process of TinyC^2

the application programming interface of the target instrumentation platform. The generated source file then can be compiled again using the common language compiler of the runtime platform. It is the responsibility of the instrumentation platform to integrate the generated aspect system and the component program together. That process is illustrated by Figure 4.

The grammar understood by ANTLR is very similar to the extended BNF grammar rules with additional manipulation options that can be defined together with the grammar. Therefore, during the evaluation process of the grammar against a source code file, a large number of customized tasks can be carried out by ANTLR to perform specific analysis tasks regarding the target language, such as tree walking, code translation, and many others.

The TinyC^2 compiler is entirely composed in Java. The most fundamental component of the translator is the `Snippet` class which is the abstraction of the generated code for a particular language element in TinyC^2. The extended or subtypes of the abstract class `Snippet` provide concrete code translation for a specific code instrumentation platform. As the parser finishes parsing the entire source code file, a parse tree is built consisting of various levels of snippets. A hierarchy of snippet objects corresponds to the structure of the source program which is defined by a finite set of grammar rules. The creation of a snippet hierarchy is illustrated by the following example.

In TinyC^2, the following rule defines the conditional `if` statement.

```
statement: . . . . . .
   | TK_if LPAREN iexpr RPAREN statement
     (TK_else statement)?
iexpr: ID  (GRT|LET)  expr
```

Figure 5: Grammar definition for if statement

The rule in Figure 5 defines that an if statement consists of a token "if" followed by "(" (`LPAREN`), then by an inequality expression, the token ")" (`RPAREN`), and a statement. The product of the rule itself is also a statement. The `iexpr` rule defines that the inequality statement is in the form of an identifier followed by either ">" (`GRT`) or "<" (`LET`) symbol, and then by a compound expression. Although those rules are indifferent from the C grammar rules, ANTLR allows us to directly place program code to get executed when a matching of the rule occurs during parsing. The code is then placed verbatim in the generated parser code. Figure 6 is the same rule given above embellished with Java code.

```
statement                                        1
returns [Snippet s = null]                       2
{Snippet subexpr, ifexpr,elsexpr;}               3
| TK_if LPAREN subexpr=iexpr                     4
      RPAREN ifexpr=statement                    5
      {                                          6
        s = new DyninstSnippet("if");            7
        s.addSnippet(subexpr);                   8
        s.addSnippet(ifexpr);                    9
      }                                          10
  (TK_else elsexpr=statement                     11
      {s.addSnippet(elsexpr);}                   12
   )?                                            13
```

Figure 6: Defining parsing behavior for if statement

To look at code example in Figure 6 more closely, line 2-3 instructs ANTLR to generate and to return a `Snippet` class for `statement` after finding a matching of the `statement` rule. Line 3 declares three sub-snippets that a snippet for the `if` statement consists of: the snippet for the condition statement, the snippet for the code block of the `if` branch, and the snippet for the code block of the `else` branch. Line 6-9 is the inserted code to actually create the snippet object of type `if` which knows how to generate the code for `if` statements. The three snippets representing the three parts of `if` blocks are inserted into the `if` snippet at line 8, 9 and 12. For example, to parse the statement: `if(a>b) b = b * a;`, a hierarchy of Snippet objects are built as illustrated in Figure 7. The left of the figure is the parse tree of the `if` statement. On the right is the image of the composition for the `Snippet` representation. Each box is the boundary of a Snippet object. The label denotes the type of the `Snippet`.

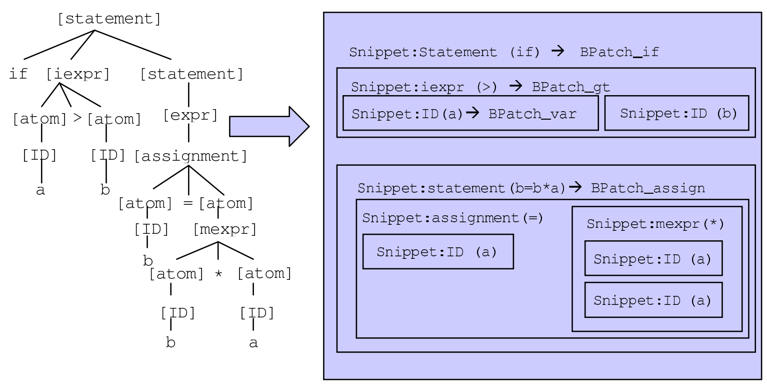

Figure 7: Snippet construction

And the symbol in the bracket represents the token(s) that the corresponding `Snippet` is responsible to translate into the target language. The labels following the arrows are the actual class types in the target instrumentation API that each corresponding snippet is translated to.

The design of the `Snippet` follows the Composite [11] architectural pattern since a complex grammar rule can be treated as a composite of the basic rules. To provide specific code translation functionality, extensions to the abstract `Snippet` class need to be defined. A concrete `Snippet` class extends the abstract method `getCode` to generate the actual target code for the corresponding language segment. In our current implementation, the translation from the TinyC^2 code to Dyninst API is carried out by the DyninstSnippet class. The code generation is initiated by invoking `getCode` at the top of the `Snippet` hierarchy which is the outer most box in Figure 7. The invocation then recursively traverses through all snippet classes after the parsing is finished. Although the instances of the DyninstSnippet class are created directly in the parser, it is easy to decouple the generated parser from any knowledge of the concrete Snippet class by using a Factory [11]. Therefore, the TinyC^2 compiler can be made backend independent by seamlessly switching to any other specialized Snippet class at class loading time.

3.3 Dynamic Weaving Mechanism

The current implementation of the backend code generator is targeted at the Dyninst runtime instrumentation platform. Therefore, the TinyC^2 code is firstly translated into C++ code in the Dyninst library API. The translated code is then compiled by a regular C++ compiler to generate a binary executable which is linked to the Dyninst instrumentation library. The executable is started with the process information of the target running system. The Dyninst library is responsible for properly inserting the code into the address space of the target program. The insertion mechanism is based on system services used by debuggers. Detailed information on how Dyninst works can be found in [7].

Leveraging the dynamic instrumentation capability of Dyn inst, TinyC^2 can be classified as a dynamic weaving aspect language. The language can be used to perform traditional non-functional activities such as tracing and performance analysis. Moreover, benefiting from the modularization capability of the language, it is convenient to develop, to maintain, and to evolve sophisticated aspect programs to intentionally change the runtime behavior of the system in a systematic manner. To understand the applicability of the dynamic weaving aspect languages, we present three case studies of the language in the following section.

4. CASE STUDIES OF TINYC^2

We can use dynamic-weaving aspect languages to increase the portability, the adaptability and the reusability of shared libraries. The reusability and the portability of libraries can be greatly improved by maintaining the properties of domain-independence and platform-neutrality. However, in practice, domain specific or platform specific constraints always require adaptations in either the library code or the application layer. Some of these constraints require changing the code in a crosscutting fashion and, thus, can be modeled by aspects. The problem with statically composed libraries including those built on static-weaving aspect languages is that it is not possible to pre-configure different versions of the shared code for every application domain or platform. And it is not safe to assume that the domain or platform specific runtime constraints are always properly addressed by the applications. Thus, runtime adaptation of libraries can be a very attractive feature especially for migrating code and dynamically configured systems. In this section, we present three case studies to illustrate such adaptations and the kind of problems these adaptations solve.

4.1 State Validation

Libraries are often shared among different application domains at runtime. Same results computed by the library might subject to different interpretations and different definitions of validity depending on the domain-specific computing requirement. We use an example to illustrate this scenario. Suppose that we want to develop a math library that provides a collection of functions to perform various integer related mathematical computations. For example, in mathematical or scientific applications, there can be no limitation to the operating range of integer values. However, in some particular computing domains such as our hypothetical statistical application for populations, there can possibly be some constraints regarding the operating range of integers and, thus, a negative result should trigger an application error. Since the goal of library design is generality, one must not hardcode the data validation logic into the library. One possible solution is to apply the validation code at every call site of the library functions that return integers. This causes the same checking code to scatter all over the places. The bloated code greatly degrades maintainability.

A more elegant and powerful solution is to compose the validation layer in TinyC^2 as aspect programs. This layer can be "woven" into the library dynamically in runtime as needed. This layer is unloaded when the library is linked into other applications. In TinyC^2, this runtime adaptation layer can be composed using the 7 lines of code in Figure 8.

```
onexit int retv group * :                              1
(int errorno, char * errormsg)                         2
{                                                      3
 if( retv < 0 )                                        4
 {                                                     5
   errorno=ILLEGAL_RESULT;                             6
   errormsg="Result cannot be negative";               7
 }                                                     8
}                                                      9
```

Figure 8: Domain specific validation in TinyC^2

We use the `onexit` construct to apply the validation (line 1). The `onexit` construct binds all the functions in the target system that return integers by using the wild card ("*") matching capability of the `group` keyword. The variable `retv` binds the specific return value of these functions. Line 2 binds global variables `errorno` and `errormsg` in the target system assuming the target system supports system wide error code schemes similar to the `errorno` of Solaris. The body of the `onexit` construct is very straightforward. It sets the `errorno` to the error code `ILLEGAL_RESULT` and assigns the error message in the target system.

If we save the file in t.c, we can invoke the compiler as `java tc t.c > Mutator.cpp`. The output Mutator.cpp is displayed in Figure 9. Lines 1-11 attach to the running process identified by its process name and process ID. Lines 12-14 invoke the `findGroupProcedurePoints` method to obtain the instrumentation points for all the functions that return integers. All the instrumentation points are collected in an object of type `BPatch_pointgroup`. Lines 20-25 create three variables to hold two global variables and the variable for the return value of the function. Lines 26-43 contain a `while` loop which iterates through every instrumentation points in the collection and inserts the `if` statements at these points in the address space of the target program.

This example shows that, although Dyninst API can be used directly by programmers, it is tedious to implement even a simple functionality. The program in Dyninst is considerably more complex and lengthy (24 lines) than our aspect program (7 lines) in TinyC^2. More importantly, the TinyC^2 program greatly improves the reusability and the adaptability of library code since no changes are made to both the math library and the application code.

```
#include "BPatch.h"                                        1
int main(int argc, char** argv)                            2
{                                                          3
  BPatch bpatch;                                           4
  char* name = argv[1];                                    5
  int pid = atoi(argv[2]);                                 6
  printf("Attaching to %s pid %d\\n", name, pid);          7
  BPatch_thread * appThread =                              8
  bpatch.attachProcess(name, pid);                         9
  appThread->continueExecution();                          10
  BPatch_image *appImage = appThread->getImage();          11
  BPatch_pointgroup                                        12
  *star_exit=appImage->                                    13
  findGroupProcedurePoints("*","int",BPatch_exit);         14
  if ( !star_exit || (*star_exit).size() == 0)             15
  {                                                        16
   printf("Unable to find exit point to \"*\"");           17
   exit(1);                                                18
  }                                                        19
  BPatch_variableExpr *errorno =                           20
  appImage->findVariable("errorno");                       21
  BPatch_variableExpr *errormsg =                          22
  appImage->findVariable("errormsg");                      23
  BPatch_variableExpr *retv =                              24
  appThread->malloc(*appImage->findType("int"));           25
  while((BPatch_Vector<BPatch_point*> *point=              26
   star_exit->getNextPoint())!=NULL)                       27
  {                                                        28
    appThread->insertSnippet(BPatch_arithExpr(             29
    BPatch_assign, *retv, BPatch_retExpr()),               30
    point);                                                31
    appThread->insertSnippet(BPatch_ifExpr                 32
    (BPatch_boolExpr (BPatch_lt, *retv,                    33
    BPatch_constExpr(0)),                                  34
    BPatch_arithExpr(BPatch_assign,                        35
    *errorno, BPatch_constExpr(1))),*point);               36
    appThread->insertSnippet(BPatch_ifExpr                 37
    (BPatch_boolExpr (BPatch_lt, *retv,                    38
     BPatch_constExpr(0)),BPatch_arithExpr                 39
     (BPatch_assign, *errormsg,                            40
     BPatch_constExpr("Result cannot be negative")         41
    )),*point);                                            42
  }                                                        43
  exit(1);                                                 44
}                                                          45
```

Figure 9: Mutator.cpp:A mutator program in full Dyninst API

4.2 Adaptive Character Encoding

The bit format for representing characters has evolved from ASCII-based single-byte encoding to multi-byte character encoding such as Unicode. For legacy systems built on the single-byte character encoding, processing information encoded by multi-byte character sets can produce erroneous results. There exist several solutions to support different character encodings in legacy code. One solution aims at providing a translation layer in between applications and the legacy code. Microsoft introduces MSLU[10] to handle the encoding translation between Unicode windows applications and windows 9X operating systems which do not support Unicode. A second solution relies on smart compilers to convert the character encoding. It requires re-

[10]`http://msdn.microsoft.com/msdnmag/issues/01/10/MSLU/default.aspx`

compilation of the system. For example, gcc[11] users can use the -fshort-wchar switch to generate 16-bit characters rather than the default 4-byte characters.

In a dynamic setting, both solutions fall short because they require the prior knowledge of the target platform and the pre-configuration of the system before the application can run. During runtime, a library could possibly be dynamically linked into several multi-byte applications, some use one type of encoding and some use another type. It is not possible to know what type of encoding to deal with until the application is running. In these situations, we can use TinyC^2 to compose the translation layer on top of the legacy code. This translation layer can be inserted into the library dynamically at run-time when it is needed. For illustration purposes, suppose in our hypothetical library, which only supports ASCII encoding, there is a group of functions which are responsible for maintaining a global message buffer. To prevent unpredicted results, our adaptation layer should first convert the characters in the buffer from a foreign encoding to the native encoding before the buffer is processed. After the buffer is processed, the adaptation layer should convert the buffer back to its original encoding. This pre/post processing logic can be implemented by the onentry and onexit constructs of TinyC^2. Figure 10 shows the TinyC^2 code.

```
onentry group buffer_ : (char * buffer)        1
{                                              2
 convert_encoding(buffer);                     3
}                                              4
                                               5
onexit group buffer_ : (char * buffer)         6
{                                              7
 restore_encoding(buffer);                     8
}                                              9
```

Figure 10: Encoding adaptation layer

This TinyC^2 code uses the group keyword to match all functions prefixed by buffer_. The onentry block (lines 2-4) invokes an external function convert_encoding which is responsible for converting the buffer into the native encoding. The onexit block (line 8) calls another external function restore_encoding to restore the original encoding. The TinyC^2 compiler generates the following code in Dyninst API.

```
#include "BPatch.h"                                          1
int main(int argc, char** argv)                              2
{                                                            3
  BPatch_thread * appThread =                                4
  bpatch.attachProcess(name, pid);                           5
  appThread->continueExecution();                            6
  BPatch_image *appImage = appThread->getImage();            7
  BPatch_pointgroup                                          8
  *buffer__entry=appImage->                                  9
  findGroupProcedurePoints("buffer_","void",                 10
  BPatch_entry);                                             11
  BPatch_pointgroup                                          12
      *buffer__exit=appImage->                               13
  findGroupProcedurePoints("buffer_","void",                 14
  BPatch_exit);                                              15
  BPatch_variableExpr *buffer =                              16
  appImage->findVariable("buffer");                          17
  BPatch_function *convert_encodingptr =                     18
  appImage->findFunction("convert_encoding");                19
  BPatch_Vector<BPatch_snippet *>                            20
  convert_encoding_args;                                     21
  convert_encoding_args.push_back(buffer);                   22
  BPatch_funcCallExpr convert_encoding                       23
  (*convert_encodingptr, convert_encoding_args);             24
  BPatch_function *restore_encodingptr =                     25
  appImage->findFunction("restore_encoding");                26
  BPatch_Vector<BPatch_snippet *>                            27
  restore_encoding_args;                                     28
  restore_encoding_args.push_back(buffer);                   29
  BPatch_funcCallExpr restore_encoding                       30
  (*restore_encodingptr, restore_encoding_args);             31
  while((BPatch_Vector<BPatch_point*> *point=                32
   buffer__entry->getNextPoint())!=NULL)                     33
  {                                                          34
     appThread->                                             35
     insertSnippet(convert_encoding,*point);                 36
  }                                                          37
  while((BPatch_Vector<BPatch_point*> *point=                38
   buffer__exit->getNextPoint())!=NULL)                      39
  {                                                          40
     appThread->                                             41
     insertSnippet(convert_exit,*point);                     42
  }                                                          43
  exit(1);                                                   44
}                                                            45
```

Generated encoding adaptation layer in Dyninst API

In the generated code, lines 4-15 attach to the running process and obtain two groups of instrumentation points, one being the entry points of all function prefixed by buffer_, the other their exit points. Lines 16-31 bind to the global message buffer and set up the function calls to convert _encoding and restore_encoding. The onentry and onexit constructs in Figure 10 are translated to two loops which insert the function calls at corresponding instrumentation points of every function in the group (lines 32-45).

4.3 Adaptive Systematic Behavior

A dynamic weaving aspect language allows us to modularize systematic properties and to build systems that are more adaptive and more efficient for specific runtime conditions. For example, middleware systems are software substrates that provide abstractions for the distributed computing entities. In a environment such as mobile computing where the platform resources and computation requirements change dynamically, it is highly desirable to configure a right set of middleware characteristics during runtime. Such high level of configurability and adaptability is hard to achieve due to non-modularized systematic properties. A typical systematic property is *Thread Safeness*. It is important for middleware systems to ensure the accesses to shared data are synchronized. However, synchronization is not always necessary for a smaller platform such as handheld devices where the underlying OS might only support a single-thread execution model due to power and memory constraints. Some middleware implementations such as TAO uses techniques

[11]http://gcc.gnu.org/

such as strategic locking [2] to allow fine tuning of locking schemes. These implementations suffer from performance overhead of redundant locking and unlocking if deployed on small platforms where the contention of resources should be minimized or avoided. The dynamic behaviors of applications such as the migration of services require middleware to load and unload properties such as Thread Safeness during runtime. A dynamic weaving aspect language such as TinyC^2 can help us achieve these goals.

To illustrate the TinyC^2 approach, suppose that the function `Service` is responsible for sending a buffer of characters to a remote entity. To ensure a valid read, the function acquires the buffer lock by invoking `lock_buffer` function before sending. It releases the lock by invoking the `release_buffer` function. Figure 11 presents the simple implementation in C.

```
int Service(char **buffer, int size)
{
   int ret = 0;
   lock_buffer();
   ret=network_send(socketfd,buffer, size);
   release_buffer();
   return ret;
}
```

Figure 11: A synchronized buffer send

As we have discussed, statically configured systems including statically weaving aspect implementations incur runtime overhead if locking is not necessary. We now provide the TinyC^2 implementation using the `onentry` and the `onexit` constructs in Figure 12.

```
onentry Service(char** buffer, int size)
{
   lock_buffer();
   //perform other operations such as checking
   //the buffer size
}

onexit Service(char ** buffer, int size)
{
   lock_release();
   //perform necessary post invocation checkings
}
```

Figure 12: TinyC^2 approach to thread safeness

The TinyC^2 compiler generates Dyninst API code in Figure 13. Similar to the previous example, lines 4-8 attach to the target process. Lines 9-14 locate the entry point and the exit point of the function `Service`. Lines 15-19 locate the function `lock_buffer` insert the function to the entry point of `Service`. Lines 20-25 load the function `release_buffer` and insert it to the exit point of `Service`. TinyC^2 does not require the functions used in the aspect program such as `lock_buffer` also defined in the component program. These functions can be compiled into a dynamically shared library and linked at runtime.

```
#include "BPatch.h"                                     1
int main(int argc, char** argv)                         2
{                                                       3
  BPatch bpatch;                                        4
  BPatch_thread * appThread =                           5
     bpatch.attachProcess(name, pid);                   6
  appThread->continueExecution();                       7
  BPatch_image *appImage = appThread->getImage();       8
  BPatch_Vector<BPatch_point*> *Service_entry=          9
  appImage->                                            10
  findProcedurePoint("Service",BPatch_entry);           11
  BPatch_Vector<BPatch_point*> *Service_exit=           12
  appImage->                                            13
  findProcedurePoint("Service",BPatch_exit);            14
  BPatch_function *lock_bufferptr =                     15
  appImage->findFunction("lock_buffer");                16
  BPatch_funcCallExpr lock_buffer(*lock_bufferptr);     17
  appThread->insertSnippet(lock_buffer,                 18
  *Service_entry);                                      19
  BPatch_function *release_bufferptr =                  20
  appImage->findFunction("lock_release");               21
  BPatch_funcCallExpr release_buffer                    22
  (*release_bufferptr);                                 23
  appThread->insertSnippet(release_buffer,              24
  *Service_exit);                                       25
}                                                       26
                                                        27
```

Figure 13: TinyC^2 approach to thread safeness

Again, our TinyC^2 implementation achieves considerable code reduction from 25 lines to 8 lines. More importantly, the synchronization facilities can be dynamically plugged in and out depending on the runtime requirements. Saving redundant locking and unlocking greatly improves the efficiency of the system.

5. RUNTIME CHARACTERISTICS OF ASPECT PROGRAMS USING DYNINST API

In this section, we examine the runtime characteristics of the application and aspect programs using addition instructions as an experiment. We are interested in two types of behaviors: 1. the "weaving" cost which is the time taken to insert the aspect code into the component program; 2. the runtime cost which is the time of computation in the dynamically inserted aspect program versus a statically written component program. We first measure the code patching cost of Dyninst. It is measured as the time taken to insert a number of "add" instructions in the target program. To measure the runtime execution overhead, we first measure the execution time of executing an increasing number of addition instructions in the component program. We then measure the same computation in the inserted aspect program. The data is collected on a Pentium IV 2GHz Linux workstation.

5.1 Code Patching Cost

Figure 14 shows the time to insert the snippet versus the number of additions in the snippet. As the size of the snippet increases, the weaving time of snippet increases rapidly. Dyninst uses the same operating system services such as ptrace and /proc file system to communicate between the application process and the mutator process. The instrumentation code is stored in large arrays which are loaded into the application process. The arrays are used for dynamically allocating small regions of memory: one is used for instrumentation variables; the other is to hold instrumentation code. A bigger snippet occupies a larger space in the array in the application memory space. It takes longer to fetch data from a larger memory space.

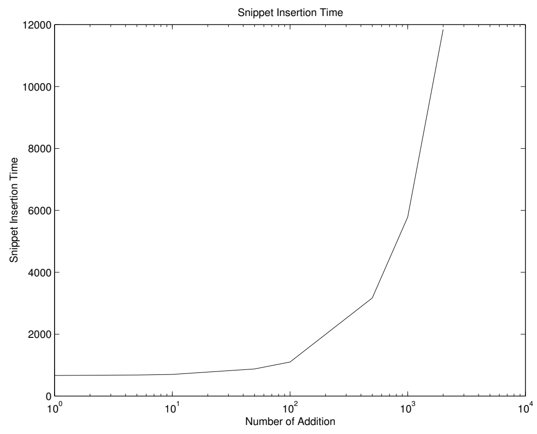

Figure 14: Code patching cost

5.2 Runtime Cost

Another important factor for dynamic weaving aspect language is the execution overhead of the aspect language as compared to carrying the same computation task in the component program. Figure 15 plots the runtime cost of performing additions in the regular C programs and in the inserted TinyC^2 code.

The running time for the same number of additions in the aspect program is significantly longer than in the component program. This can be explained by the runtime instrumentation mechanism of Dyninst. The original code in the application process branches into newly generated code through use of `trampolines` [7]. Trampolines are short sections of code that provide a way of getting from the point to the newly generated snippet. Several steps are involved here. Firstly, one or more instructions at the instrumentation point are replaced with a branch to the start of a base trampoline. Then the base trampoline code branches to a mini-trampoline. The mini-trampoline saves the current machine state and contains the code for a single snippet. At the end of the single snippet, code is placed to restore the machine state and to branch back to the base trampoline. The base trampoline executes the original instruction(s) in the application code. Therefore, there is significant management overhead for executing the aspect program in the case of Dyninst. Another reason is that since the aspect code is inserted during runtime, the code misses the static compiler optimization stage and, therefore, produces un-optimized code.

5.3 Limitations and Open Questions

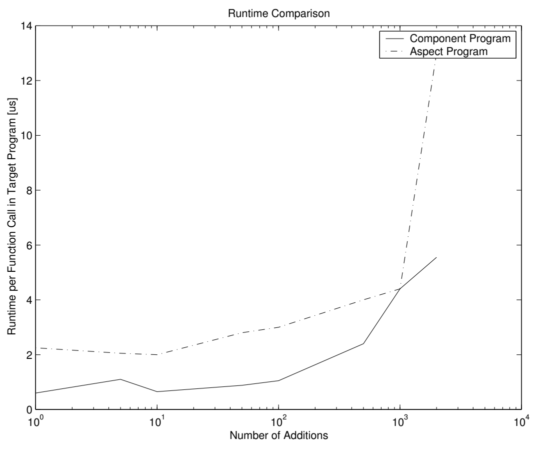

Figure 15: Runtime cost of TinyC^2 code versus regular C code

There are many limitations of the current implementation of the TinyC^2 language. Firstly, the language is being implemented as a prototype. We hope to demonstrate its capability of implementing large scale and complex aspect oriented systems by our continuous extension of the language. The second limitation comes from the limitations of Dyninst. The API of Dyninst was not designed to support aspect languages. Features such as modifying function arguments and their return values are not yet possible to implement using Dyninst. We have added a number of APIs to Dyninst to support the "group" language construct.

There are also many challenges regarding implementing dynamic weaving aspect oriented systems in general. The first category of challenges is comprised of performance related issues of dynamically woven AOP systems. Our experimental data show that the cost of computing in dynamically inserted code is considerably high. One reason is that dynamically inserted code misses the optimization stage in the compilation process which leads to un-optimized code. Intuitively, advanced compiler techniques such as dynamic optimization techniques [10] can be used to further optimized the mutated code during runtime. However, there are several issues regarding dynamic optimization. Firstly, from a compiler point of view, the newly patched code might disturb any optimization strategy that the compiler has chosen for the code. Runtime code patching can also trigger subsequent runtime optimization, which adds a considerable overhead to the overall runtime cost. Secondly, it is not clear to us if the runtime optimized code still allows us to detach the inserted aspect code on the fly as part of the dynamic adaptation. A third prominent issue is that current aspect language designs require preservations of weaving points, e.g. function identifiers in the context of the TinyC^2 language, in order for weaving to work. This is a trivial concern for static-weaving languages. However, these identifiers in the source code might disappear in the runtime code due to compiler optimization techniques such as code specialization, function inlining, and many others. Certain identifiers or symbols must be made available to aspect weavers at all time. But does the preservation of symbols decrease the optimization gain? Is there a measure of such trade-offs?

The second category of challenges concerns designing dynamic weaving languages is that whether there should be language facilities to take advantage of its dynamic nature. For example, Dyninst gives us some degree of control over the running state of the target program during the code patching process. Should the design of a dynamic weaving language gives first-status concerns to issues such as controlling the state of the target program, runtime information of the platform, optimization related tasks, and many others?

The third category of challenges includes issues regarding the security of dynamic weaving languages. That is the dynamically inserted code must comply with the security policies of the target platform. These policies could include execution privileges and copyright protections.

6. CONCLUSION

In this paper, we presented the work of TinyC^2, an aspect oriented language that is designed to syntactically extend the C programming language and to use existing code instrumentation platforms as the backend. A prototype of the language compiler is developed to support a subset of the standard C language features with a couple of additional language constructs. The backend instrumentation platform is provided by Dyninst runtime instrumentation platform.

Through this work, we demonstrate the possibility of supporting certain aspect oriented language semantics by using code instrumentation platforms. We prove the concept that code instrumentation techniques and the aspect oriented design goals are fundamentally compatible as one can be used to express the other. A language approach in bridging the two domains is viable because, as illustrated in the case study, we are able to express higher-level programming concerns in the form of TinyC^2 language and to realize those concerns through the form of code instrumentation.

It is currently not possible to have a complete evaluation of the language approach presented in this paper, since the full aspect language features are still needed to be developed. We also need to experiment with a different instrumentation tool to verify if the consistency of the language semantics can be maintained. Finally, from the experience of this work, we have encountered several issues regarding the viability of the runtime weaving aspect language design. These issues are mainly concerned with the cost of dynamically changing the runtime behavior of the system. We expect further research on advanced AOP compilers will develop solutions to these problems.

Acknowledgements

We are very grateful to Michael J. Voss who pointed out to us the similarities between runtime instrumentation techniques and aspect oriented mechanisms. The initial performance analysis and the graphs in this paper are prepared by Yiqian Ying.

7. REFERENCES

[1] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, 2002.

[2] Douglas Schmidt Michael Stal Hans Rohnert Frank Bushmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.

[3] Morgan Deters Ron K. Cytron. Introduction of Program Instrumentation using Aspects. *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, pages 131–147, 2001.

[4] A. Srivastava A. Edwards and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report Technical Report MSR-TR2001 -50, Microsoft Research, One Microsoft Way, Redmond,WA, April 2001.

[5] B. Grant M. Philipose M. Mock C. Chambers S.J. Eggers. An Evaluation of Staged Run-time Optimizations in DyC. *Conference on Programming Language Design and Implementation*, May 1999.

[6] M. Arnold S. Fink D. Grove M. Hind and P.F. Sweeney. Adaptive Optimization in the Jalapeno JVM. *Object-Oriented Programming Systems, Languages and Applications*, 2000.

[7] Bryan Buck Jeffrey K. Hollingsworth. An API for runtime code patching. *Journal of Supercomputing Applications and High Performance Computing.*

[8] Charles Zhang Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. *International Conference of Aspect Oriented Software and Development*, pages 130–139, 2003.

[9] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[10] Bala Vasanth Duesterwald Evelyn Banerjia Sanjeev. Transparent dynamic optimization. Technical Report HPL-1999-77, Hewlett Packard, 1999.

[11] Erich Gamma Richard Helm Ralph Johnson John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[12] Jeffrey K. Hollingsworth Barton P. Miller Marcelo J. R. Goncalves Oscar Naim Zhichen Xu and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques*, 1997.