

# COMS E6998-3 Term Project Report

## *Advanced Topics in Programming Languages and Compilers, Fall 2012*

### MIPLex: Adapting Dynamic Code Modification to the MIPL Language

YoungHoon Jung  
Dept. of Computer Science  
Columbia University  
New York, NY, 10027  
Email: jung@cs.columbia.edu

#### I. INTRODUCTION

In this project, we extend MIPL, a Prolog-compatible programming language with distributed computational features, to evaluate how adapting a dynamic code modification technique will affect the extended programming language, MIPLex, focusing on the execution performance and the program development convenience of the language. The experiments will shed a light on the possible benefits of dynamic code modification in languages targeted at Java Virtual Machines, from the performance and the development convenience perspectives.

##### A. Contributions

In order to address the possibility of utilizing dynamic code modification in programming languages that have a Java Virtual Machine (JVM)-based backend, we developed MIPLex and experimented the extended MIPLex compiler. The main contribution of this project is to provide the proof of concept that testifies to the benefits of using the dynamic code modification technique for compiler implementations. In doing so, we first propose frontend expansions of the language to offer to its users a way to benefit from the dynamic nature of these expansions. Second, we implement a representative subset of the proposed frontend expansions using the dynamic code modification technique. Finally, it is also intended in this project to understand how this concept plays a role from the performance perspective so that we possibly get the idea to which direction this concepts can be utilized further.

##### B. Scope

The scope of this project includes the development and the extension of the language syntax (frontend) as well as the target language generation part (backend). On top of that, experiments will be conducted to figure out whether these extensions help to gain the performance improvement or the ease of development.

We pursue, by comparing the results between executions using the implemented extensions and the ones without

them, the insights how the dynamic code modification feature can be utilized for a various purposes in the language designs or compiler implementations and the future direction of this project.

##### C. The MIPL language

MIPL (Mining-Integrated Programming Language) is originally designed for large matrix computations through automated matrices and their computations distribution over a cluster through the MapReduce framework.

Its syntax is compatible to Prolog so that any Prolog programmers can also easily write a MIPL program, with a bit of learning for the MIPL *job* that supports distributed matrix operations. A *job* is a function-like sub routine that can return multiple number of return values and facts are converted to matrices when they are given to a *job* as arguments.

For the types, it adopts a dynamic and weak typing system so that the MIPL programmer can write a MIPL program easily and quickly.

##### D. Code Repository

As the git repository used when the MIPL language and its compiler has been developed [2], the MIPLex compiler and materials for its experiments is developed on and uploaded to a public git repository [1], hosted by github.com. The repository can be accessed by a git command, “git clone <https://github.com/jcybha/MIPLex.git>”.

#### II. PROJECT PROCESS

##### A. Milestones

**Performance Improvement.** As dynamic code modification may bring performance improvements to the certain programming languages, a grammatical extension will be devised to adopt dynamic code modification for initialization, status changes, and matrix loading. Experiments will verify that these adaptations for each case actually bring performance increases.

**Development Support.** For the better development experiences to the users, dynamic code debugging suggestion will be proposed, developed, and verified.

## B. Action Plan

The process of this project has followed the action plan in Table 1.

| Date     | Plan  |
|----------|---|
| ~ Oct 23 | Project Initialization<br>Repository Creation   |
| ~ Oct 24 | Proposal  |
| ~ Oct 31 | Performance Improvement<br>- <i>Front/Backend Design</i><br>- <i>Implementation &amp; Experiments</i> |
| ~ Nov 09 | Development Support<br>- <i>Feature Design</i><br>- <i>Implementation &amp; Experiments</i>           |
| ~ Nov 13 | Demo Scenario & Presentation  |
| ~ Nov 20 | Experiments & Analysis  |
| ~ Dec 6  | Final Report  |

Table 1. Time-lined Action Plan

## III. MOTIVATIONS

Since many algorithms need operations like making variables proper initial values and doing one-time computations, a feature supported by the language will help its programmers conveniently write programs with this operations. In particular, it is very common for programmers to use a boolean variable and an if-clause to handle initializations or one-time computations in a loop or a function called repeatedly. For example, a Java programmer may use the boolean-and-if solution as shown in Listing 1.

Listing 1: The boolean-and-if Solution

```
boolean initialized = false;
...
if (!initialized) {
    ...
    initialized = true;
}
...
```

This solution can be applied to MIPL codes in the same principle. However, this solution causes performance degradation when this code is in a loop as the example in Listing 2. Even though the evaluation of the if-clause is required only once, the loop makes the evaluation happen at every iteration.

Listing 2: An Example of the boolean-and-if Solution in a Loop

```
job some_algorithm(A, B, C, D, E) {
    I = 0.
    while (I < 1000) {
        .
        A = B + C.
        if (I == 0)
```

```
        A *= D.
        A += E.
        .
        I += 1.
    }
}
```

One can avoid the performance degradation of the boolean-and-if solution by splitting the first iteration from the loop as show in Listing 3.

Listing 3: An Example of Manual Optimization for a Loop

```
job some_algorithm(A, B, C, D, E) {
    I = 0.
    .
    A = B + C.
    A *= D.
    A += E.
    .
    while (I < 999) {
        .
        A = B + C.
        A += E.
        .
        I += 1.
    }
}
```

Again, despite the fact that this manual splitting overcomes the performance degradation it significantly decreases developability, readability, or maintainability of the code since basically it duplicates the code in the loop.

## IV. THE NEEDS FOR LANGUAGE EXTENSIONS

As described in Section III, there is a need for a feature that solves the performance degradation problem without decreasing engineering efficiency, i.e. readability, developability, and maintainability. The code in Listing 4 shows how the keyword *once* solve the problem without aforementioned disadvantages.

Listing 4: An Example of the Keyword *once*

```
job some_algorithm(A, B, C, D, E) {
    I = 0.
    while (I < 1000) {
        .
        A = B + C.
        once {
            A *= D.
        }
        A += E.
        .
        I += 1.
    }
```

```

    }
}

```

## V. SYNTAX EXTENSION FOR DYNAMIC CODE MODIFICATION

### A. *once* Keyword for One-Time Executions

Here, we introduce the keyword *once* for the one-time execution of the statement<sup>1</sup> that follows the *once* keyword. There are two distinct flavor of the *once* keyword; *Anonymous Once* and *Named Once*.

1) *Anonymous Once*: *Anonymous Once*, or *Spot-Scope Once*, is a feature that allows programmers to make a location-bound code so that the specific code will be executed only once throughout the process lifetime. This type of *once* keyword is distinguished by its location in the source code only, in other words each of *once* keywords is always considered distinctly. Thus, the one-time execution of a statement with *once* keyword is never affected by other *once* keywords.

Listing 5 shows two jobs, each of which contains a *once* keyword. In this example, there are two *Anonymous Onces*. During the first call of the *job* 'do\_some\_job' the statement with the *once* keyword will be executed. From the second call, the statement will not be executed as the *once* keyword is designed. Then, if the *job* 'do\_another\_job' is called, the sentence associated with the *once* keyword in the *job* is executed once only at the first iteration of the loop during the first call. In other words, for the rest iterations of the loop or during the subsequent calls, the sentence is ignored.

Listing 5: An Example Use of *Anonymous Once*

```

job do_some_job(A, B, C) {
    once {
        do some initialization.
    }
    do rest jobs.
}

job do_another_job(A, B, C) {
    while (A < 100) once do some.
    do rest jobs.
}

```

2) *Named Once*: *Named Once*, or *Global-Scope Once*, is a feature that allows programmers to make a name-bound code so that only one of the codes associated with the given name will be executed once throughout the process lifetime.

The example in Listing 6 presents three *Named Onces*, two of which are associated with the same name, "Init A" while the other *Named Once* has a distinct name, "Init B". If the 'do\_some\_job' *job* is called first, both initializations

under the two *once* keywords will be executed. After then, if the *job* 'do\_another\_job' is called, the initialization under the *once* keyword in the *job* will not be executed because the name "Init A" is considered already executed.

Listing 6: An Example Use of *Named Once*

```

job do_some_job(A, B, C) {
    once ("Init A") {
        do some initialization.
    }
    once ("Init B") {
        do some initialization.
    }
}

job do_another_job(A, B, C) {
    once ("Init A") {
        do some initialization.
    }
}

```

### B. Generalizations of *Once* Keyword

Since the *once* keyword is a specialized feature that executes a specific code at a specific condition, two possible generalization can be developed from the *once* keyword; the *state* and *times* keywords.

1) *state Keyword*: The *state* keyword allows a programmer to conveniently build a finite state machine (FSM). Like the *once* keyword is a solution for a boolean-and-if implementation, the *state* keyword works as a int-and-switch implementation, which uses an integer variable to store the state and checks the state and execute statement according to the state value. Similarly, the *state* keyword; i) executes the statement associated to the particular state is executed, and ii) updates the state value as designated after the statement.

Listing 7: An Example Use of *state* Keyword

```

job do_some_job(A, B, C) {
    status (Named_state) ["init_state"] {
        some work.
    } ["state1"]
    ["state1"] {
        some work.
    } ["state2"]
}

job do_another_job(A, B, C) {
    status (Named_state) ["state2"] {
        some work.
        next_state = "state2".
        if (cond_for_state3)
            next_state = "state3".
    } [next_state]
}

```

<sup>1</sup>The term *statement* in MIPLex is used identically to the same term *statement* in the C language

```

    ["state3"] {
        some work.
    } ["finish"]
}

```

2) *times Keyword*: The *times* keyword is another generalization of the *once* keyword. Instead of executing the statement once, the *times* keyword takes a positive integer  $n$  to executed the statement for the first  $n$  times. In the example shown in Listing 8, the statement associated to the *times* keyword will be executed for the first 10 times.

Listing 8: An Example Use of *times* Keyword

```

job do_some_job(A, B, C) {
    times (10) {
        statement.
    }
}

```

### C. Synchronization

The semantics of each keyword assume synchronization. For instance, the definition of the *once* keyword allows only first reach to the *once* will be actually executed. There are two synchronization issues with regard to its semantic; i) if two or more thread are reaching to the same *once*, only one of them will be execute the associated statement, and ii) while once thread is executing the *once*, the other threads should be blocked until the thread finishes the execution of the *once*.

## VI. IMPLEMENTATION ISSUES

In this Section, we address the challenges and four different approaches in implementing the compiler’s backend for the proposed *once* keyword using the dynamic code modification technique. The key issue that occurs in the implementation of the *once* keyword is that a JVM does not support the code modification in the memory but we try to achieve the same effects using roundabout ways, i.e. creating a new class and loading the class.

Since the MIPLex language’s backend generates Java Bytecode that runs on JVMs, the implementations discussed in this Section is about Java Bytecode. The examples in this Section, however, will be presented in Java source codes to help readers understand.

### A. Impl-1: Naïve Approach

The easiest way to implement the *once* keyword is using a boolean and an if clause as Java programmers do III, as shown in Listing 9. This approach ensures the simplest compiler backend implementation, however as mentioned it degrade the performance since the conditional branch is evaluated in every iteration of a loop or in a *job* call.

Listing 9: Naïve Approach for *once*

```

class SomeClass {
    boolean initialized = false;
    void someMethod(.) {
        if (!initialized) {
            do some initialization;
            initialized = true;
        }
        rest of operations;
    }
}

```

### B. Impl-2: Two Classes Approach

To reduce the performance degradation from the approach in Section VI-A, this approach generates two class files, one with the condition checking and the statement to execute once and another one without them as shown in Listing 10. This approach follows these steps: i) the implementation class with the conditional statement is loaded as other usual class, ii) after the conditional statement is executed, it loads the target class with the second implemented class without the conditional statement, and replace the original class with newly loaded class, and iii) when the *job* is called again, now the method in the newly implemented class will be called instead of the method in the first class.

Listing 10: Two-class Approach for *once*

```

class SomeClass {
    void someMethod(.) {
        do some initialization;
        rest of operations;
        reload this class;
    }
}

class SomeClassWOInit {
    void someMethod(.) {
        rest of operations;
    }
}

```

This approach is, however, impractical for the implementation because it statically generate the class files during the compile time. Since a class can have more than one *once* keywords in a class file, some of which may be considered executed previously if they are *Named Once* and a *once* with the same name could have been executed in another method. For handling each combination, this approach will at compile time create  $2^k$  class files where  $k$  is the number of distinct *once* keywords in the class file.

### C. Impl-3: Dynamic Code Modification Approach

This approach solves the aforementioned problems by adopting the dynamic code modification technique, which

generates modified class files at runtime. The compiler generates class files from jobs with the *once* keywords and thus need to be modified at runtime, with i) the initialization code, ii) the modification code, and iii) the class reloading code, as shown in Listing 11. When this code is executed, the initialization is executed once and it creates a new class based on this class, but removing the instructions from the initialization code to the class reloading code, as illustrated in Listing 12. Finally, the class reloading code is executed, replacing the class instance with an instance of the new class so that the initialization, the modification, and the class reloading code will not be executed again, from the next execution of the *job*.

Listing 11: Dynamic Code Modification - Old Class

```
class SomeClass {
    void someMethod(.) {
        do some initialization;
        modify codes;
        reload affected classes;

        rest of operations;
    }
}
```

Listing 12: Dynamic Code Modification - New Class

```
class SomeClass {
    void someMethod(.) {
        rest of operations;
    }
}
```

This approach, however, still have a drawback; even after the class reloading, the code currently being executed will not be changed, which we call a *non-immediate effectiveness* problem.

**Non-immediate Effectiveness.** As the old code currently being executed in the memory remains the same even after the compiled code reloads the class and replaces the class instance, continuation of the execution may result in the unintended consequences. Two possible problematic examples are presented in Listing 13 and 14.

Listing 13: An Example of Non-immediate Effectiveness

```
job a_job() {
    I = 0.
    while (I < 100) {
        once {
            initialization.
        }
        I = I + 1.
    }
}
```

Listing 14: Another Example of Non-immediate Effectiveness

```
job a_job() {
    once (.A.) {
        initialization.
    }
    once (.A.) {
        initialization.
    }
}
```

#### D. Impl-4: Dynamic Code Modification Approach with Immediate Effectiveness

To resolve the *non-immediate effectiveness* problem, this approach adopts an intermediate step to correctly complete the first execution of the job. This approach is based on the previous approach in Section VI-C. In addition, this approach i) generates an intermediate class in Listing 16, which is basically the same class with the newly created class in Listing 17, but with a feature that reveals its own code address in the memory. After the old class finishes the class reloading, it jumps to the corresponding offset of the intermediate class. In this way it can successfully completes its first execution with an immediate effectiveness of the class modification.

Listing 15: Immediate Effectiveness - Old Class

```
class SomeClass {
    void someMethod(.) {
        do some initialization;
        modify codes;
        reload affected classes;
        get the address by call;
        jump to the rest part;
        rest of operations;
    }
}
```

Listing 16: Immediate Effectiveness - Intermediate Class

```
class SomeClass {
    void someMethod(.) {
        get address;
        return;

        rest of operations;
    }
}
```

Listing 17: Immediate Effectiveness - New Class

```
class SomeClass {
    void someMethod(.) {
```

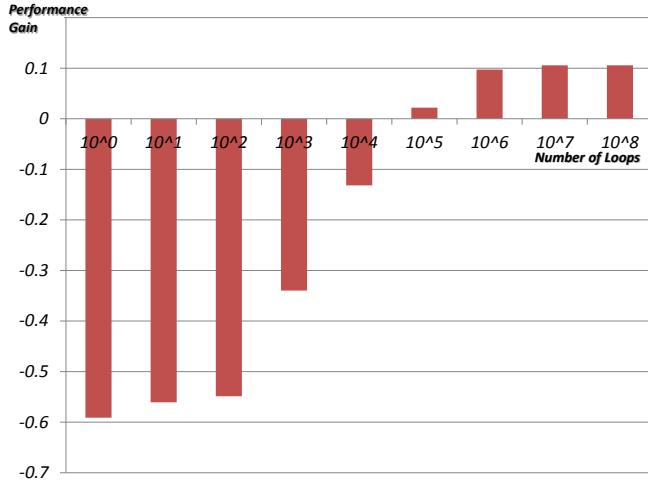


Figure 1. Performance Gain from Impl-1 over Impl-4.

```

    rest of operations;
}
}

```

**Address Manipulation.** Getting an address of a code is restricted in a JVM by design. However, there is only one possible chance that a JVM reveals a code address. *JSR* is a Java Bytecode instruction that jumps to the target label and places the address of the target label. By inserting a *JSR* instruction and its target right after the *JSR* to the beginning of the intermediated class, the old class can get the corresponding target code address in the memory. Likewise, the old class jumps to the target code address with the *RET* Java Bytecode instruction, which continues execution from address taken from a local variable.

## VII. PROACTIVE DEBUGGING

Along with the *once* keyword family, the proactive debugging support is also one possible use case that utilizes the dynamic code modification technique for the compiler’s backend. With the proactive debugging feature, the compiled code can suggest the possible modification of the compiled code when a runtime error occurs, for the debugging purposes. If the programmer accepts the modification suggestion, the code modifies itself so that from the next execution the code will follow the fixed routine.

For example, if a MIPLex program tries to multiply two matrices, both of which has a dimension of  $(m \times n)$ , it first shows an Non-compatible Matrix Size Error and suggests the user to change the code to an addition or a subtraction. If the user accepts the suggestion, the class file is modified accordingly. Applicable MIPLex runtime errors include: Non-compatible Matrix Size Error, Non-matched Number of Return Variables Error, Operand Type Mismatches, and so on.

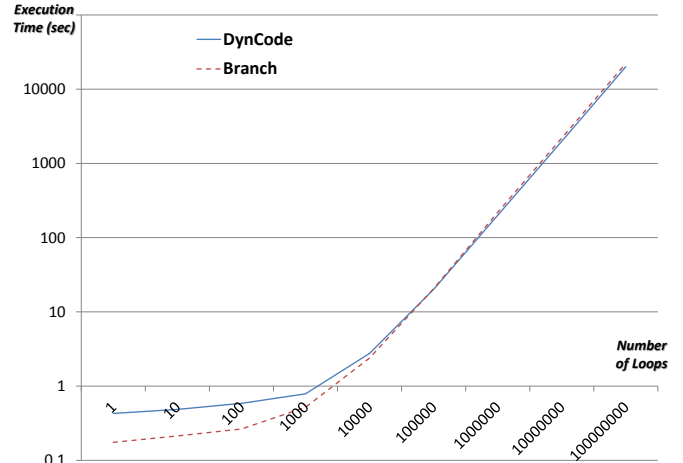


Figure 2. Execution Time Comparison between Impl-1 and Impl-4.

## VIII. EXPERIMENTS

In this section, we evaluate the performance benefit from utilizing the dynamic code modification technique to implement the *once* keyword to the MIPLex language. To compare the implementations, we refer in this section the implementation of the naive approach in Section VI-A as Impl-1 and the implementation of the dynamic code modification with immediate effectiveness in Section VI-D as Impl-4.

### A. Performance Gain

Figure 1 shows that the performance gain (X-axis) from Impl-4 over Impl-1 with regard to the number of iterations in a loop (Y-axis) that contains the *once* keyword. The performance gain grows from -60% to 10% as the number of loops increase. Note that the performance gain is below zero for the execution of the loops less than  $10^4$  due to the overheads from the class modification and the class reloading. The performance gain increases and converges approximately to 10% as the number of loop increases, compensating the overheads from the dynamic code modification.

### B. Execution Time Comparison

Figure 2 is the plotting of the execution time of Impl-1 and Impl-4 with regard to the number of loops. Both lines grow proportionally to the number of loops, with Impl-4 slightly less inclined. This means that Impl-4 shows better performance as the *once* keyword is more repeated in the loops, which effectively hides the overheads of the dynamic code modification.

## IX. CONCLUSIONS

In this project, we proposed a language extension, implemented the compiler backend, and experimented the compiled code, in order to evaluate the feasibility and benefits of utilizing the dynamic code modification technique.

The extended language feature, through the keyword *once*, provided a convenient notion for the one-time execution of a specified code. Its implementation in the compiler backend actualized a feasible solution of the concept. The experimental results showed the proposed extension and the implemented compiler backend can bring an advantage over the naïve implementation in terms of the performance.

#### REFERENCES

- [1] “MIPL (<https://github.com/jinhyung/mipl>).”
- [2] “MIPLex (<https://github.com/jcybha/mipllex>).”