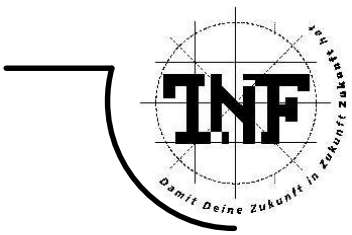
JOHANNES KEPLER
UNIVERSITY LINZ
Research and teaching network



Jakob Praher

A Change Framework based on the Low Level Virtual Machine Compiler Infrastructure

THESIS

in partial satisfaction of the requirements for the degree of

DIPLOM-INGENIEUR
in the Master Program Computer Science

Created at the *Institute for System Software*
Johannes Kepler University Linz

Supervised by:

o.Univ.-Prof. Dipl.-Ing. Dr. Dr. h.c. Hanspeter Mössenböck

Linz, April 2007

Abstract

When developing or deploying large applications, one would like to have more insights into what an application is doing at runtime. Frequently it is required to change defective parts of an application as fast as possible. For instance one may wish to replace a certain function call in a program with another function call whenever a specified condition holds. This master thesis aims at building the change framework, a system for dynamic program instrumentation and analysis. This research builds atop of the Low Level Virtual Machine (LLVM) for representing C/C++ applications in an intermediate form. The change framework consists of two parts, the application under analysis, and a monitor process. The application under analysis is a C/C++ application compiled to LLVM bytecodes. The monitor process communicates with the application process and is able to dynamically instrument and analyze the application process using a domain specific language. This change language has powerful constructs for defining and conditionally applying application changes. An important overall goal of this system is to ease the analysis as well as alteration of low level system software at run-time.

Kurzfassung

Häufig während der Entwicklung und im Einsatz komplexer Anwendungen möchte man mehr Informationen und Einblicke haben, was eine Anwendung zur Laufzeit macht. Oft ist es auch notwending fehlerhafte Teile schnell auszutauschen. Zum Beispiel das Ersetzen einer Funktion mit einer anderen Funktion, immer wenn eine bestimmte Bedingung eintritt. Diese Diplomarbeit beschäfigt sich mit dem Change Framework - ein System zur dynamischen Analyse und Instrumentierung von Programmen. Die Arbeit verwendet die Low Level Virtuelle Machine (LLVM) zur Repräsentation von C/C++ in einem Zwischenformat. Das Change Framework besteht aus zwei Teilen. Einerseits der Anwendung, die analysiert werden möchte und andererseits einem Monitorprozess. Erstere ist eine Anwendung, die auf LLVM Bytecodes übersetzt ist. Der Monitorprozess kommuniziert mit dem Anwendungsprozess und nimmt Änderungen an der Anwendung zu deren Laufzeit vor. Diese Änderungen werden in einer domänspezifischen Sprache, der Change Language, verfasst. Diese Sprache bietet mächtige Konstrukte zur Definition und bedingten Anwendung von Änderungen an Applikationen. Eines der Hauptziele dieses Systems ist es die Analyse und die Veränderung von Anwendungen zur Laufzeit zu vereinfachen.

Contents

List of Figures

Chapter 1

Introduction

Most of todays computers are based on the so called *von Neumann Architecture*. A program is stored in memory as a consecutive block of instructions. The program is executed by fetching an instruction from memory, decoding it to detect its type, executing it, and writing the results back to memory. The instruction format and layout determines the computer architecture or instruction set architecture (ISA).

Writing programs in this way is very difficult and error prone. Over the years, programming languages have been created to make writing computer programs easier. In order for the processor to execute a program written in a programming language, it has to be transformed into a block of instructions. Debugging information maps source language artefacts to instruction regions. This is essential to map runtime errors of a program to the originating source language constructs. This information is very expensive. Typically one would strip it off as soon as the program is not used for debugging anymore.

When developing or deploying large applications, one would like to have more insights into what an application is doing at run-time. For instance one may wish to replace a certain function call in a program with another function call whenever a specified condition holds. This kind of analysis and transformation is hard to achieve with ahead-of-time compilation models. Recompilation is often needed when changing code. The running process has to be stopped, and new machine code executable has to be created and executed.

An approach often used for generating log information is to add logging code at various places and define the level of verbosity when launching an application. This is more laborintensive and the many unused logging instructions can cause significant run-time overhead. An ideal approach would be to dynamically change specific areas of a program based on specified criteria. This master thesis describes an environment for this purpose.

The work presented here is based on the LLVM (Low Level Virtual Machine) infrastructure. LLVM allows one to compile source code written in C/C++ to a low level intermediate representation. This interemediate representation can then be ahead-of-time compiled to the target machine code or dynamically executed using a so called execution engine.

1.1 Dynamic Program Instrumentation and Analysis

According to the thesaurus[1] *analysis*, can be defined as *examination*. Additionally to examination, program analysis is the task of understanding the behavior and the nature of a program. While static analysis refers to analyzing a program's instructions or a program's source code, dynamic analysis refers to gathering information about a program during its execution. The term *at run-time* is often used to describe the time during that a program is executed. This is an important distinction. Due to the conditional nature of computer programs, the *same program* can yield *different results* depending on its execution context.

An important part of program analysis is *instrumentation*. The online encyclopedia Wikipedia defines[2] *instrumentation* as:

> *Instrumentation*is defined as "the art and science of measurement and control". Instrumentation can be used to refer to the field in which instrument technicians and engineers work, or it can refer to the available methods of measurement and control and the instruments which facillitate this.

Program instrumentation means to change the program's instructions in order to analyze the program. In computer science, the term instrumentation is used when changing programs by adding and removing code with the concern of analysis, control, or measurement of the instrumented code. This is a *spearate concern* from the application code that solves a problem in the application domain. Often instrumentation code gets inserted after the application code has been written.

As with static program analysis, static program instrumentation refers to adding instrumentation code before executing the program. Dynamic or interactive instrumentation on the other hand means adding and removing instrumentation instructions at run-time. An important distinction is that dynamic program analysis can also be done with static instrumentation.

Understanding a complex software especially in case of wrong behavior or collecting important performance data implies instrumentation of the right parts of the application code. Often it is impossible to anticipate run-time behavior during static analysis of a program. There is also a trend in modern optimizing compilers. Some compilers use profiling data collected by previous runs of the application in order to help static performance optimizations.

On the other hand modern just-in-time compilers rely on so called hot-spot information which is collected at run-time. Dynamic instrumention allows the developer/user to change parts of a program on an as needed basis. This greatly reduces the complexity of the instrumated code.

[1] `http://thesaurus.reference.com/search?q=analysis`
[2] `http://en.wikipedia.org/wiki/Instrumentation`

1.2 Scenarios for Dynamic Program Analysis

The best way to see the importance of program instrumentation and analysis is by example. Applications of dynamic program analysis are manyfold. On a macrolevel there is for instance the big topic of finding errors in large applications. Origins of errors are often hard to track only by looking at the static structure of the program. Below is an example of a typical problem of a big application with many possible run-time configurations.

Observed Problem. After a certain amount of successful HTTP requests a webserver suddenly freezes and stops responding. Restarting the webserver results in new requests being successfully processed. After a number of requests the same effect occurs again. The log files do not contain any error descriptions.

Real Cause of Problem. Each request is logged using reverse DNS resolution to obtain the address name of the requesting user agent. The webserver blocks while resolving the address in question. Due to a problem in the DNS resolution settings, all child processes of the webserver are blocked by the DNS resolver. The webserver is unable to process any more requests and freezes.

Possible Analysis.

- Trace the main loop of the webserver process.
- Write all functions called by the webserver onto standard output.
- The output shows that the resolver library is blocking.
- Examine the call stack to see where the application is calling the resolver library.
- It shows that the log subsystem calls the DNS resolver.
- Look through the configuration subsystem to find that the logging format includes name resolution.
- Disable name resolution for log entries and continue.
- Determine the name resolution problem. Solve the problem there and turn on reverse name lookup again (if needed).

Furthermore on the microlevel, dynamic instrumentation can be used to optimize certain application areas. Typical applications follow an 80-20 rule. At least 80 percent of the run-time of an application is spent in at most 20 percent of the code. Static optimization can not easily determine these hot-spots. Additionally, hot-spots can vary, depending on the run-time context. In this case, instrumentation and analysis can help to determine hot-spots. With dynamic instrumentation it is also possible to change individual functions to better optimized ones without stopping the program execution.

Chapter 2

System Architecture

2.1 A Framework for Program Analysis

This master thesis aims at building a system for dynamic program instrumentation and analysis. Figure 2.1 shows the architecture. It consists of two processes:

- The application process
- A monitor process

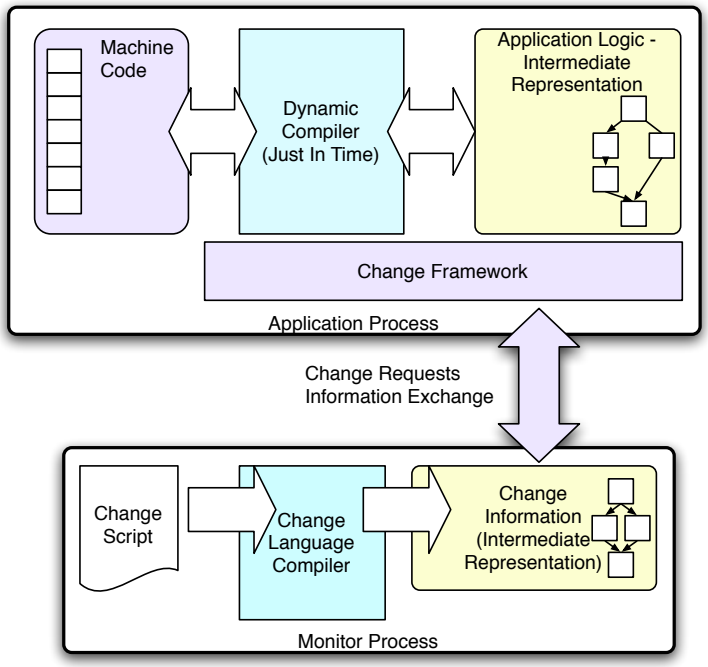


Figure 2.1: Core Architecture for Program Analysis

The *application process* contains the application program to be instrumented and analyzed. Instead of just being a native executable, the application process contains the application logic in an *intermediate representation* (IR). This IR is dynamically compiled to native code using a *just-in-time (JIT) compiler*.

The application process also contains the *change framework*. The change framework uses information available from the application's IR to instrument the code. It listens for requests from the external *monitor process* and processes these. So called *changes* are requests to instrument and change the application code. Changes are applied by instrumenting the IR. After that, the framework informs the dynamic compiler to recompile the changed application code. All applied changes are logged by the framewok. An applied change can be unapplied again. This enables interactive usage and incremental analysis.

The second process in Figure 2.1 is the *monitor process*. A monitor process communicates with an application process over means of interprocess communication (IPC). Its main task is to compile changes to IR and to communicate with the application's process. Changes are written in a specific language, the *change language*. As can be seen in Figure 2.1, the monitor process includes a compiler to translate change language scripts. To be easily incorporated into the application's IR, they are compiled to the same IR. After the change scripts are compiled, they can be transmitted to the application process for getting processed by the change framework.

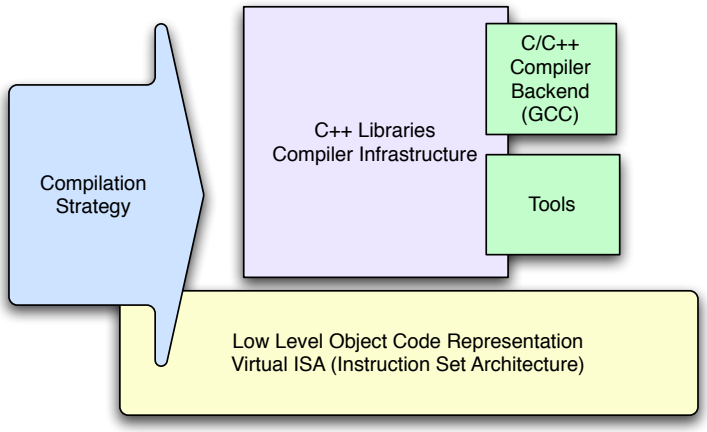


Figure 2.2: LLVM Components

The IR in use is the intermediate representation of the Low Level Virtual Machine (LLVM) [LLV06e] compiler infrastructure [LA04]. LLVM is both an infrastructure as well as a framework for compiler engineering. As can be seen in Figure 2.2 LLVM is:

- *A compilation strategy*: Instead of doing translation-unit at a time compilation, the LLVM strategy is to have whole program representations. One application is continously optimized by staged compilation techniques during the run time of a program.
- *A low level object code representation* building a virtual instruction set architecture (V-ISA).
- *A compiler infrastructure* made of C++ class libraries.
- *Various tools* for compiling, assembling, linking, debugging, and introspecting LLVM bytecode files.
- *GNU Compiler Collection (GCC) backend*: Instead of generating native code out of C/C++ source files, this version of GCC produces LLVM bytecodes.

These characterstics made LLVM a suitable foundation for this research.

2.2 LLVM Intermediate Representation

Representing the application logic in LLVM IR eases the gathering of static and dynamic information. Additionally, changes can be made in a more portable and safer way than if the framework would instrument machine code. The information contained in the LLVM IR is the main information model available to the change framework. As can be seen in Figure 2.3 the LLVM IR has the following structure:

- A module: The module is the container of functions and global variables.
- Functions: Functions are named, callable units of instructions.
- Global variables: Global variables are values that are accessible by all functions.

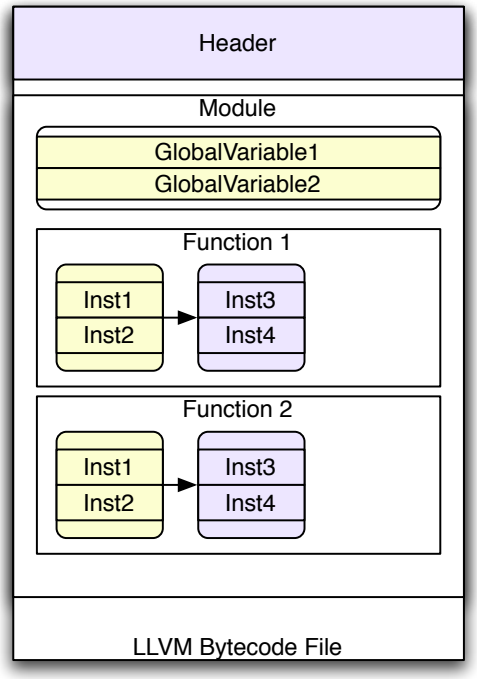


Figure 2.3: LLVM Bytecode Model

As discussed in Chapter 4 the LLVM IR is not object oriented and resembles a C like model albeit with higher level type information. In this view the function is the first class way of representing logic. Functions further consist of a list of basic blocks, each basic block consists of a set of instructions.

A basic block is a sequence of instructions that always execute sequentially without changing the control flow. Because of that property they play an important role in compiler optimization algorithms. Since LLVM is used as a framework for compiler engineering, basic blocks have a first class representation. In this overview we just see basic blocks as additional structural information.

Additionally LLVM provides the following advanced information:

- Def-Use information
- CFG (Control flow graph) based on basic blocks and terminator instructions
- Data flow information based on the Static Single Assignment (SSA) form [CFR+91]
- High-level type information (primitives, pointers, arrays, structures, functions)

This information is used by the change framework when instrumenting applications. The more information that is available the better the analysis of a program can be performed. As with basic blocks, most of the concepts listed above are used in algorithms of optimizing compiler passes. The LLVM concepts will be discussed in more detail in Chapter 4 starting on page 25.

Another property of the LLVM IR is that all the information is available at no additional costs. This information gets persisted in LLVM bytecode files. LLVM's intermediate representation is principally used as an in memory representation. Furthermore it can be serialized into a binary bytecode format [LLV06a]. This format is designed in a way that keeps most information without the need for expensive recomputation.

2.3 Change Framework Overview

The change framework applies changes to a running application. After the IR gets changed the function is recompiled. Each transformation is captured by a change definition. Figure 2.4 shows that changes are defined using the change language. It is a C like language with specific support for changes.

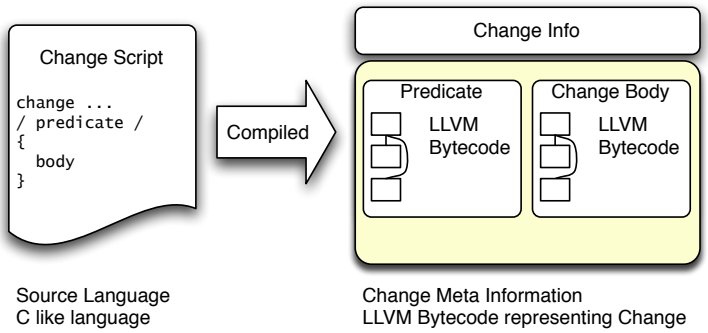


Figure 2.4: Details of the Change Model

A key concept of the change framework are *change providers*. Change providers operate inside the application's process. They are responsible for:

- Collecting information and exporting information to changes.
- Checking change predicates and inserting change bodies at well defined change points.

A change provider can have a certain granularity. The granularity is the scope the provider is working on:

- Per module: This means the provider is only interested in modules, and does not look into the details of a module.
- Per function: This means the provider is interested in functions.
- Per basic block: This means the provider operates on basic blocks.
- Per instruction: This is the finest granularity. The provider operates on each instruction.

Providers collect information and apply changes. Provider definitions build the interface between the provider and the change scripts. The exported information is provider specific.

```
function provider functionProvider {

  /* Valid Change Points for this provider */
  points { OnEnter, OnLeave };

  int id;                /* instance id */
  string name;           /* name of the function */
  int numArgs;           /* number of arguments */
  bool isVarArg;         /* is it a variable argument function */

  /* .. */
};
```

Listing 2.1: Change Provider Defintion

Listing 2.1 shows an example function provider. The granularity is the first modifier of the provider definition. This provider would have `function` granularity. It's name is `functionProvider`. It gets invoked for every function in the LLVM bytecodes. Furthermore it is able to insert changes at `OnEnter` and `OnLeave` of a function. The field declarations `id`, `name` represent exported information.

Providers are like plugins. Everybody comfortable with the LLVM class libraries and bytecode format can write his own provider. Every provider that is declared like in Listing 2.1 can be used in change scripts.

Provider definitions as in Listing 2.1 make providers available to change scripts. A change definition, like in Figure 2.2, uses one or more providers with the intent to change parts of an application. This is done by using the information made available by the provider. A change definition consists of:

- *A principal provider*: The provider that is responsible for transforming the bytecode and applying the change.
- *A change point*: A point, as exported by the provider, that describes the location where the change is to be inserted.
- *A change predicate*: A predicate function that uses information exported by available providers. The change predicate determines *if* a change applies.
- *A change body*: The actual transformation. This is the code to be inserted at the change point if the predicate evaluates to `true`.

The principal provider is responsible for applying the changes. The change point depends on the provider chosen. For instance a change using our example `functionProvider`

provider can choose the change points `OnEnter` or `OnLeave`. Provider writers are responsible for choosing self-explanatory names for change points.

```
change functionProvider::OnEnter
/ functionProvider->name == "main" /
{
  /* actions to be performed */
}
```

Listing 2.2: A Change Definition

Change predicates are declared between two slashes (`//`). A change predicate tests a possible change point for the specified predicate condition.

```
/ functionProvider->name == "main" /
```

Listing 2.3: A Predicate using the Provider `functionProvider`

The predicate shown in Listing 2.3 is invoked by the function provider once for every function in the application. If the predicate evaluates to `true` for a function f, the change body gets inserted at the specified change point in function f. Change predicates are tested during change application. They do not cause any runtime overhead. The change body represents the change itself. The change language supports a subset of C statements and expressions. In the example above the actions get inserted when entering a function called `main`.

By convention every provider definition exports an *implicit* constant value. This implicit value is usable from within a change definition that uses that provider. The example change definition shown in Listing2.2 uses the provider `functionProvider`. This enables the change to access the constant value `functionProvider`, that represents a pointer to the fields exported by that provider. `functionProvider->name` refers to the current function's name as exported by the provider Listing 2.1.

2.4 Compiling Applications to LLVM

Applications need to be compiled to LLVM bytecode to be used with the change framework. This section gives an overview of traditional machine code compilation as well as the strategies to compile LLVM bytecode. When compiling source code to machine code, individual source files, so called translation units, are compiled, assembled and linked into an executable or library.

Figure 2.5 shows this model. Each translation unit is compiled into a separate object file. The object files are linked together by the linker (link editor) to produce a single executable or library file. The resulting executable is suitable for direct execution by the underlying operating system.

Compiling to LLVM bytecodes is different to compiling native code as showed in Figure 2.5. The native code is executable by the operating system. When targeting the LLVM architecture this is not the case. The underlying operating system and processor do not understand the virtual instruction set. Figure 2.6 shows the two strategies for the

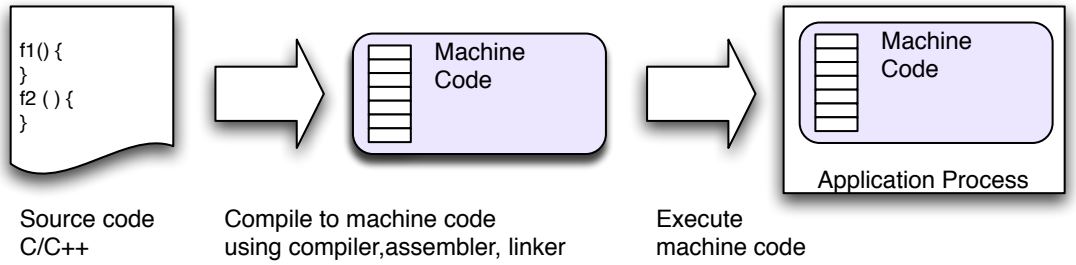


Figure 2.5: Compiling and Executing C/C++ to Machine Code

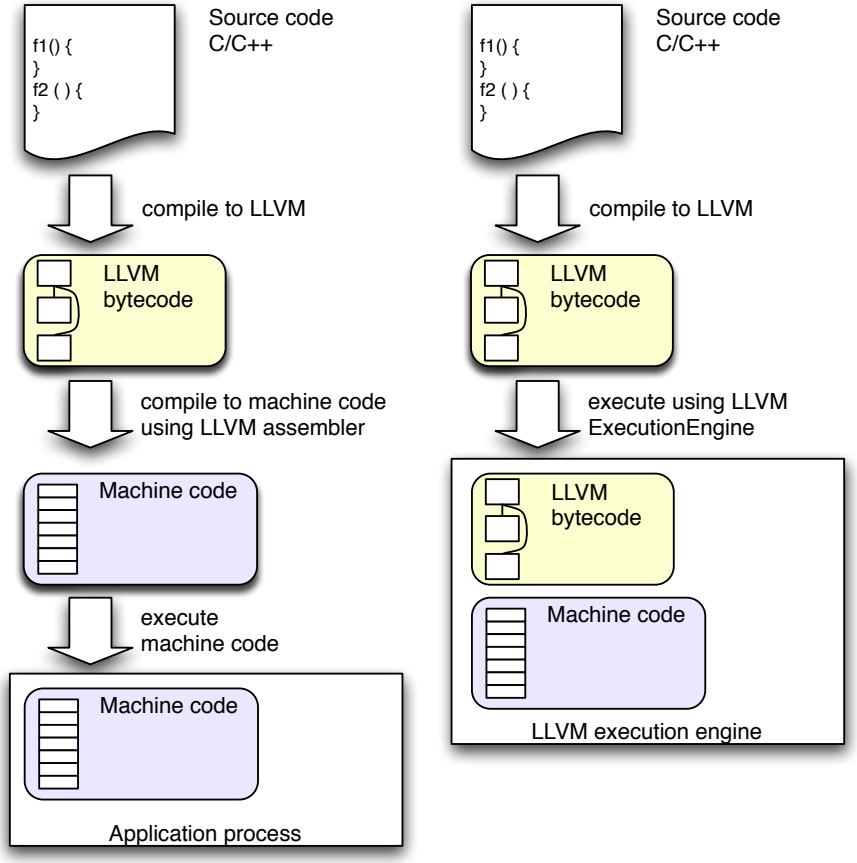


Figure 2.6: Executing C/C++ via LLVM

execution of LLVM bytecodes. On the left side C/C++ source code gets translated into LLVM bytecodes. These bytecodes are then compiled to machine code using ahead-of-time compilation. Compiling source languages this way differs only slightly from the one showed in Figure 2.5. The most important difference is that this model leads to better whole program optimizations.

On the right side, the bytecodes are dynamically compiled to machine code on an as needed basis. Instead of directly loading and executing the native executable, a host application, called an execution engine, is loaded. This execution engine in turn loads the application's bytecodes. For this research the dynamic compilation strategy was

used. In this approach the code is dynamically compiled on an as needed basis and the compilation subsystem is already in the application's process. While this is more suitable for dynamic instrumentation, a static compilation strategy would also work.

2.5 Recompiling Instrumented Program Fragments

The last piece in the system architecture deals with recompiling instrumented program fragments. When a change request arrives in the application process, the change framework processes it. Changes are applied by transforming the applications IR, which leaves the machine code version untouched at first. The application process' machine code needs a way to detect that there is a modified version of its intermediate representation available, so that it recompiles the modified parts of the system.

In order to make the generated machine code aware whenever it should recompile itself, a *check code* is injected in the application's IR at load time. The change framework keeps a global flag that represents the dirtiness of the bytecodes. If the bytecodes have been changed, the dirty flag gets set to true. After checking the dirtiness the flag is reset, so that the next bytecode change can be done right after a check was performed, not after the machine code was recompiled.

The dirty state check is performed before a function is called. This is since the LLVM execution JIT compiler translates on a per function granularity. A more detailed explanation of the recompilation algorithm can be read in Chapter 5 beginning on page 45.

2.6 Summary

The aim of this chapter was to provide the reader with an essential overview of the system architecture. To summarize, important properties of the change framework are:

- *Dynamic*: Changes can be applied at run-time.
- *Reversible*: Changes can be unapplied again.
- *Extensible*: Providers export instrumentation functionality.
- *Interactive*: A set of change application/unapplication can be used to learn more about the running program. Information output is transferred from the Execution Engine to the change monitor process.
- *Portable*: Changes are applied by transforming LLVM IR. Providers are thus portable to any processor architecture, targeted by the LLVM platform.
- *Network transparent*: The change framework and the change monitor are different processes. Communication is performed through interprocess communication (IPC) which need not take place on the same host. Since change scripts are compiled to LLVM bytecode, the change framework is not dependent on the system architecture of the monitor process.

Chapter 3

Big Picture

3.1 Overview

In the last chapter the system architecture of the change framework was introduced. As shown in Figure 2.1 the framework consists of two processes: The application to be observed, called the *application process*, and the *monitor process*, that generates change requests out of change scripts. The monitor process is the tool the user works with in order to manipulate the application process at run-time.

Change providers run inside the application process and provide information, so called change points as well as export information about the application process. Change scripts use change providers and transform the application process' intermediate representation. The change framework operates using bytecode transformation. Any function, or variable that is used by an application and is available in LLVM bytecode form can be analyzed and transformed.

This chapter provides motivating examples on using the change framework. The next section starts by defining a reference source example in the C programming language. This will build the foundation for all change framework examples throughout the chapter.

3.2 Dynamic Program Analysis

This section provides scenarios for using the change framework. The example code fragment shown in Listing 3.1 forms the basis for change scripts in this section. The result of the change script is presented as displayed in the monitor program. Two examples for porgam analysis will be given. First information about the function call sequence and loop detection is shown. Then memory usage information will be analyzed. The examples are written in the change language and provide use cases for the change framework as defined in Chapter 5.

3.2.1 Example Code Fragment

This subsection provides the example code fragment that serves as a foundation for the change scripts that will be shown in the next sections.

```
// externally defined
void do_work_a() ;

void do_work_b() {
  puts("An important message");
  do_work_a();
  do_work_a();
}

bool selector_func() {
  static int counter = 0;
  counter ++;
  return counter % 2;
}

void complex_function() {

  while (true)
  {
    if ( selector_func() ) {
      for (int j = 0; j < 2 ; j++ )
      {
        do_work_a();
      }
    }
    else {
      do_work_b();
    }
  }

}
```

Listing 3.1: Example Code Fragment

Listing 3.1 shows the example code fragment, the function `complex_function` shows a loop that performs calls and `do_work_b` depending of run-time state. The function `do_work_a` is externally defined, while `do_work_b` prints a message on the standard output stream and calls `do_work_a` two times.

3.2.2 Program Analysis

Many interesting questions can be asked about code fragments like the one showed in Listing 3.1. For instance one could ask the following questions:

- What is the function call sequence?
- Is there a loop inside the function?
- How many bytes of heap memory are allocated, freed?

The answers to this questions are given in the next sections. It is assumed that the application has been compiled to LLVM bytecodes and the application is executed using the change framework. In order to get these information, the application is instrumented by change scripts that get applied to the application process using a monitor process. Theses change scripts are written the change language that is part of the change framework.

All the information is gathered by the monitor process at run-time. The changes could be applied to any application running within the change framework.

3.2.3 Function Call Sequence

Analyzing the function call sequence using the change framework is done using a simple provider that operates on a per function granularity. The provider is exported as `functionProvider` below:

```
function provider functionProvider {
  points {OnEnter, OnLeave};

  int id;
  string functionName;
  int argCount;
  int isVarArg;
};
```

For more details on change providers please refer to Section 5.3. Providers can be used in a change script, like the following:

```
change functionProvider::OnEnter
//
{
  io->printf( "==> Entering function '%s'\n", functionProvider->
    functionName );
}
```

Listing 3.2: A Change Script to Print Out the Name of Every Called Function

To apply the above change script, that resides in the file `function.cl` to a process with the id 22300, the monitor application `monitor` could be executed as follows:

```
monitor register 22300 function.cl
```

The change monitor would communicate with the application via the change protocol. If successful the following would get written on the console of the monitor, depending on the time of the registration and the semantics of the called functions:

```
==> Entering function 'selector_func'
==> Entering function 'do_work_a'
...
==> Entering function 'selector_func'
==> Entering function 'do_work_b'
==> Entering function 'puts'
...
==> Entering function 'do_work_a'
...
==> Entering function 'do_work_a'
...
==> Entering function 'selector_func'
==> Entering function 'do_work_a'
...
```

Since `do_work_a` and `puts` are defined in another translation unit, one does not know the details of them by looking at Listing 3.1, so the listings show '...' lines, which stand for *various output messages*. Besides the builtin value `io`, which is capable of sending output to the monitor's console, the change framework has another builtin value called `thread` which acts as a thread local storage. This builtin value works like a map where the change script author can store thread local information. The script of Listing 3.2 can be enhanced by making use of thread local storage:

```
change functionProvider::OnEnter
//
{
  int callLevel = thread->getInt("callLevel");
  string space = "| " * callLevel;
  io->printf( "| %s+ Entering function '%s'\n", space,
    functionProvider->functionName );
  thread->putInt("callLevel", callLevel+1);
}

change functionProvider::OnLeave
//
{
  int callLevel = thread->getInt("callLevel");
  string space = "| " * (callLevel - 1);
  io->printf( "| %s+ Leaving function '%s'\n", functionProvider->
    functionName );
  thread->putInt("callLevel", callLevel-1);
}
```

Listing 3.3: A Change Script to Print Out the Name of Every Called Function

After un-registering the old change script and registering the new change script the following should appear on the monitor console, again depending on the run-time properties:

```
| + Entering function 'selector_func'
| + Leaving function 'selector_func'
| + Entering function 'do_work_a'
| |  + ...
| + Leaving function 'do_work_a'
| |  + ...
| + Entering function 'selector_func'
| + Leaving function 'selector_func'
| + Entering function 'do_work_b'
| | + Entering function 'puts'
| | |  + ...
| | + Leaving function 'puts'
| | + Entering functino 'do_work_a'
| | |  + ...
| | + Leaving function 'do_work_a'
| + Leaving function 'do_work_b'
```

Listing 3.4: Function Call Sequence with Call Nesting Level

By using a single provider on a per function granularity, and two builtin objects, `thread` and `io`, a function call tree of the application is printed. Depending on the exposed information of the provider much more information could get printed out, for example the type of arguments or the return type.

The current change scripts do not have any change predicates, indicated by an empty `/ /` change predicate. If one would be only interested in a certain class of functions, the change script could be changed like in Listing 3.5:

```
0 change functionProvider::OnEnter
  / functionProvider->functionName[0..7] == "do_work" /
2 {
    int callLevel = thread->getInt("callLevel");
4   string space = "| " * callLevel;
    io->printf( "| %s+ Entering function '%s'\n", space,
      functionProvider->functionName );
6   thread->putInt("callLevel", callLevel+1);
  }
8
  change functionProvider::OnLeave
10 / functionProvider->functionName[0..7] == "do_work" /
  {
12   int callLevel = thread->getInt("callLevel");
    string space = "| " * (callLevel - 1);
14   io->printf( "| %s+ Leaving function '%s'\n", functionProvider->
      functionName );
    thread->putInt("callLevel", callLevel-1);
16 }
```

Listing 3.5: A Change Script to Print Out the Name of Every Called Function, Filtered by Function Name

The change script above, would yield the following call tree:

```
| + Entering function 'do_work_a'
| |  + ...
| + Leaving function 'do_work_a'
| |  + ...
| + Entering function 'do_work_b'
| | + Entering functino 'do_work_a'
| | |  + ...
| | + Leaving function 'do_work_a'
| + Leaving function 'do_work_b'
```

Listing 3.6: Funciton Call Sequence With Call Nesting Level

In Listing 3.5 the change predicates that filter the change points are defined on line two and eleven. In order to yield a correct result they have to be the same. Such a change predicate is evaluated for every point where the provider is applied to. The change predicate '`/ functionProvider->functionName[0..7] == "do_work" /`' can be read as: Apply this change if the first 7 characters of the `functionName` attribute matches the string "do_work". As can be seen from the source code fragment in Listing 3.1 this limits the functions to `do_work_a`, and `do_work_b`. In the change language substrings can be expressed by the sring slice `astring[a..b]`. The slice expression $a..b$, a is the lower bound and b the upper bound. The string goes from position a to $b-1$.

Additionally if the application contains other functions that also start with `do_work` and are also called, these function would be printed here too. This is due to the fact that the change framework is no source code transformer. Instead it operates on LLVM bytecode transformation. Hence any function that is part of the application and available in LLVM bytecode form is subject to transformation. All the provider information defined in the

provider declaration can be used inside change predicate expressions. The attributes of the provider `functionProvider` are defined in Listing 3.2.3.

3.2.4 Loop Detection

Another interesting question to ask about a code fragment is to print out information about loops. Loops are particularly interesting in computing since they give a hint about hot spots in a program. A hot spot in an application is a set of instructions that are executed quite frequently.

In principal there are two ways to do loop detection. One is dynamically, at run-time. This could be accomplished by tagging every basic block during transformation. And later at run-time keeping track which blocks have already been visited. If a block is visited twice inside the same function, there exists a loop. This is the dynamic method. There exists static approaches. A static approach is discussed here. This is based on information obtained from the control flow graph (CFG) of a function. As discussed in Chapter 4, LLVM provides explicit control flow information.

A CFG is a directed graph whose nodes are so called basic blocks. The edges of a CFG represent control flow from one block to another block. A basic block can be defined as a sequence of instructions that are always executed in sequence that is always entered at its first instruction and exited at its last. Subsection has more information about CFGs. [CT04] or [Muc98] provide more information about control flow data structures.

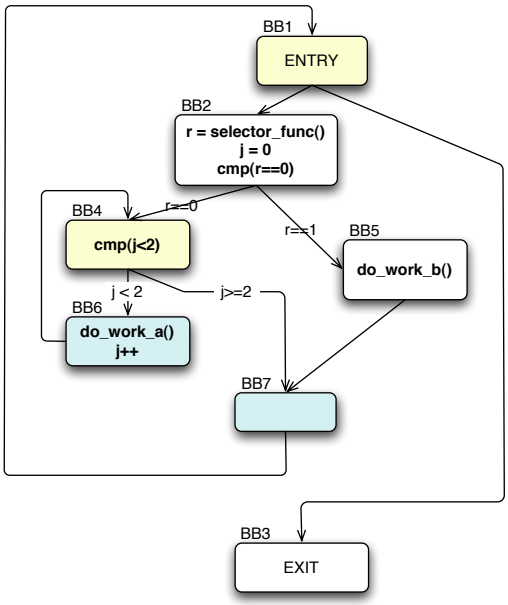


Figure 3.1: CFG of the Example Code Fragment

Figure 3.1 shows the CFG for the example code fragment in Listing 3.2.3. The yellow nodes show a loop header, a blue node shows a node which can only be reached by a loop header, but points back at the loop header as well. The CFG provides one with all the possible control-flow paths in a function. The CFG is a directed graph. Loops

in the program are cycles in the CFG. While there are many ways to do loop detection in a CFG, a very simple one is to traverse the graph using a depth first search (DFS), and to mark nodes as either one of: *univisted*, *visiting in progress*, *visited*. An edge is a looping back-edge if the destination node is already in state *in progress*. A node goes from *unvisited* to *visited* if all nodes reachable from this node are visited. While the reachables have been visited the state of the node is set to *visiting in progress*.

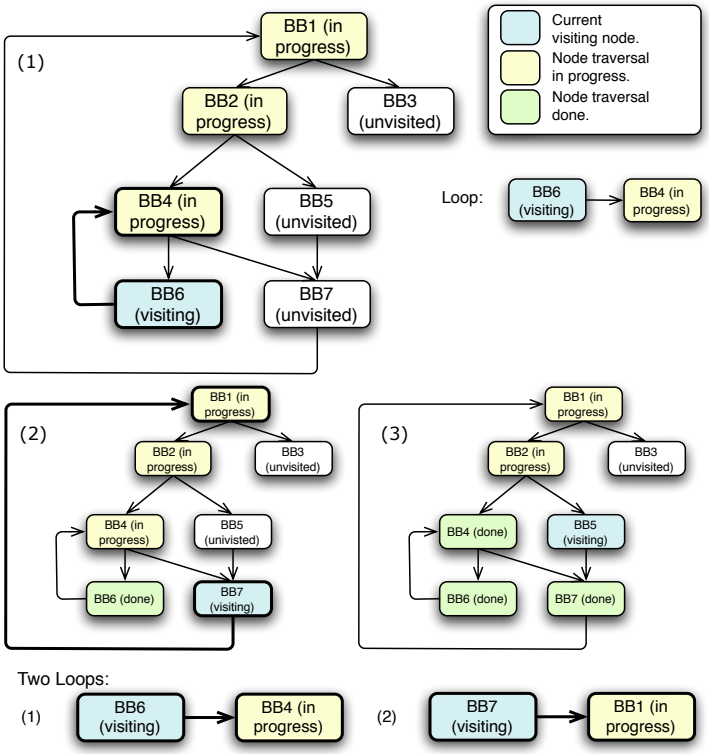


Figure 3.2: Loop Detection by Depth First Search Traversal

Figure 3.2 shows the traversal based on the CFG from Figure 3.1. There are two loops in this graph. The back-edges are from block BB_6 to block BB_4 and from block BB_7 to block BB_1. The figure shows the DFS traversal in three different states. In the first snapshot BB_6 is currently beeing processed. As BB_6 points back to BB_4 and BB_4 is still *in visiting* state, this is identified as a loop. In the second snapshot BB_7 is actively processed. Note that this is still as part of the traversal of BB_4 (BB_4 is still *in visiting* state and BB_5 is still *unvisited*. In this snapshot BB_7 points back to BB_1 and BB_1 is still *in visiting* state, this is the second loop in the CFG. The last snapshot shows that BB_5 is actively being processed. This points to BB_7 which is already marked as *visited*. After this step BB_3 has yet to be processed before the whole graph is traversed.

With this information one can construct a loop information provider. The loop information provider operates on a per function granularity and collects and exports loop information. It could look like:

```
function provider loopInfo {
  point {BeforeLoop, WithinLoopHeader, AfterLoop};
```

```
  int id;
  string functionName;
  int maxNumberOfBasicBlocks;
  int numberOfExits;

};
```

The `loopInfo` provider above offers the following change points: `BeforeLoop`, `WithinLoopHeader`, and `AfterLoop`. `BeforeLoop` means the change code will get inserted before the loop code. Hence this code will run only one time per loop. `AfterLoop` happens also only once per loop and describes a point in the application before the first code outside the loop is executed. It differs from `BeforeLoop` that `AfterLoop` code might be inserted at more than one place. Since a loop might have more than one control flow outside the loop. `WithinLoopHeader` means the code is inserted inside the loop header. This code will run each time the loop header is run.

This provider can be used to write the following simple change script:

```
change loopInfo::WithinLoopHeader
//
{
  io->printf("Loop id=0x%X detected inside function %s.\n", loopInfo->
    id, loopInfo->functionName);
}
```

This change script runs within the loop header. This means it is executed every time the loop header is executed. If it would be registered with our example application, it would print something like:

```
Loop id=0x1 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x1 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x2 detected inside function complex_function.
Loop id=0x1 detected inside function complex_function.
...
```

The second loop code is executed three times. The loop body only executes 2 times. This is because the loop header is entered 3 times. Thus the change code will be also executed three times. As an enhancement we count the number of times a loop header is executed. This is done by changing the change script to look like:

```
change loopInfo::BeforeLoop
//
{
  thread->setInt(loopInfo->id, 0);
}

change loopInfo::WithinLoopHeader
//
{
  int execCount = thread->getInt(loopInfo->id);
```

```
  io->printf("Loop id=0x%X detected inside function %s, exec count=%d
    .\n", loopInfo->id, loopInfo->functionName, execCount+1);
  thread->setInt(loopInfo->id, execCount + 1);
}
```

Re-registering the change, this should print something like:

```
Loop id=0x1 detected inside function complex_function, exec count=1.
Loop id=0x2 detected inside function complex_function, exec count=1.
Loop id=0x2 detected inside function complex_function, exec count=2.
Loop id=0x2 detected inside function complex_function, exec count=3.
Loop id=0x1 detected inside function complex_function, exec count=2.
Loop id=0x2 detected inside function complex_function, exec count=1.
Loop id=0x2 detected inside function complex_function, exec count=2.
Loop id=0x2 detected inside function complex_function, exec count=3.
Loop id=0x1 detected inside function complex_function, exec count=3.
...
```

To make output more interesting, a nesting level is added to the change output.

```
change loopInfo::BeforeLoop
//
{
  thread->setInt(loopInfo->id, 0);
  int loopNest = thread->getInt("loopNest");
  thread->setInt("loopNest", loopNest + 1);
}

change loopInfo::WithinLoopHeader
//
{
  int execCount = thread->getInt(loopInfo->id);
  string space = "| " * (thread->getInt(loopNest)-1);
  io->printf("| %s+Loop id=0x%X detected inside function %s, exec
    count=%d.\n", loopInfo->id, loopInfo->functionName, execCount+1);
  thread->setInt(loopInfo->id, execCount + 1);
}

change loopInfo::AfterLoop
//
{
  int loopNest = thread->getInt("loopNest");
  thread->setInt("loopNest", loopNest - 1);
}
```

The above change script would send the following output to the monitor process.

```
| + Loop id=0x1 detected inside function complex_function, exec count=1.
| | + Loop id=0x2 detected inside function complex_function, exec count=1.
| | + Loop id=0x2 detected inside function complex_function, exec count=2.
| | + Loop id=0x2 detected inside function complex_function, exec count=3.
| + Loop id=0x1 detected inside function complex_function, exec count=2.
| | + Loop id=0x2 detected inside function complex_function, exec count=1.
| | + Loop id=0x2 detected inside function complex_function, exec count=2.
| | + Loop id=0x2 detected inside function complex_function, exec count=3.
| + Loop id=0x1 detected inside function complex_function, exec count=3.
```

```
...
```

As can be seen the rich information represented in the LLVM intermediate format can be effectively used to gather information about a program at run-time.

3.2.5 Memory Usage

In long running applications, the use of memory, especially dynamic memory, is of interest. The LLVM intermediate language has first class support for memory allocation/freeing via specific instructions. For example the code fragment below allocates 128 bytes on the heap memory.

```
char * p = (char* ) malloc( sizeof(char) * 128 );
```

The formal description of the LLVM `malloc` and `free` instruction is

```
result = malloc <type>[, uint <numElements>][, align <alignment>]
free <type> <value>
```

The C language malloc fragment can be translated to LLVM intermediate code as:

```
%p = malloc sbyte, uint 128
```

The `malloc` instruction has a reference to the element type, as well as the number of elements allocated.

Freeing dynamically allocated memory in the C langauge is performed through the `free` function. `free` takes a pointer to the memory region to be freed. After calling `free` this memory region is not in use by the application any more.

```
char * p = (char* ) malloc( sizeof(char) * 128 );
free(p);
```

The above C language fragment can be translated to the folloing LLVM code snippet.

```
%p = malloc sbyte, uint 128
free sbyte* %p
```

In order to get information about memory allocation, a mallocInst provider that provides information about `malloc` instructions is introduced:

```
instruction provider mallocInst {
  points {Before, After};

  int id;
  string elementType;
  int elementSize;
  int numElements;
  int alignment;
};
```

Similarly for information about `free` we use the `freeInst` provider:

```
instruction provider freeInst {
  points {Before, After};

  int id;
  string type;
  string elementType;
  int numElements;
  int valueId;
};
```

For the purpose of this example we define the `do_work_a` function from our example code fragment as:

```
/*
 * Collects data from some source
 */
int read_data(char* data, int size_bytes);
/*
 * Counts the number of digits in the integer
 */
int count_digits(int d);

/*
 * Prints the size of elements in bytes
 */
char* print_size(int elements) {
  int digitCount = count_digits(elements);
  int length = digitCount + strlen("DATA:  bytes");
  char * p = (char *) malloc(sizeof(char) * (length + 1));
  *p = 0 ;
  sprintf(p, "DATA: %d bytes", elements);
  return p;
}

/*
 * Allocates a buffer and reads the data.
 * Prints information about the data.
 * Frees the data again.
 */
void do_work_a( ) {
  const int SIZE_BYTES = 1024;
  char *p = (char*) malloc(sizeof(char) * SIZE_BYTES);
  memset(p, 0, sizeof(char)*SIZE_BYTES);
  int data = read_data(p, SIZE_BYTES);
  puts(data_info(data));
  free(p);
}
```

The `do_work_a` function allocates two byte arrays on the heap. One is a buffer into which the `read_data` function writes data into. The other array is a string array created by the `print_size` function. Note that the string buffer that gets allocated in the `print_size` function does not get freed again.

The change script below prints all the allocation/free information within the function `do_work_a` using the `mallocInst` and `freeInst` providers.

```
change mallocInst::After
//
{
  io->printf("Allocated %d bytes on the heap\n", mallocInst->
    numElements * mallocInst->elementSize );
}

change freeInst::After
//
{
  io->printf("Freed %d bytes on the heap\n", freeInst->numElements *
    freeInst->elementSize );
}
```

Registering this change script with the application process should yield:

```
...
Allocated 1024 bytes on the heap.
Allocated 14 bytes on the heap.
Freed 1024 bytes on the heap.
...
```

The above output shows that the 14 bytes are not freed again. The next step is to get more information about the non-freed data. Since the providers in use are on an instruction level, the change predicates as well as the change body can make use of all providers of level basic block, function, and module. This is further explained in Chapter 5. This functionality can be used to print the function that contains the instructions using the already introduced `functionProvider`:

```
change mallocInst::After
//
{
  io->printf("[%s] Allocated %d bytes on the heap\n", functionProvider
    ->functionName, mallocInst->numElements * mallocInst->elementSize
    );
}

change freeInst::After
//
{
  io->printf("[%s] Freed %d bytes on the heap\n", functionProvider->
    functionName, freeInst->numElements * freeInst->elementSize );
}
```

Registering the above script would yield:

```
...
[do_work_a] Allocated 1024 bytes on the heap.
[print_size] Allocated 14 bytes on the heap.
[do_work_a] Freed 1024 bytes on the heap.
...
```

As can be seen above the non-freed memory was allocated in the function `print_size`. The change framework could even be used to correct a misbehavior like memory leaks in application **if** one knows what to do. The behavior above could make perfect sense, if the allocated memory is stored in some global variable or further processed by some other functions. On the other hand it could simply be a mistake. However the change framework makes diagnosing errors much easier.

Other memory related instructions of interested would be the load/store instructions. In the LLVM all memory transfers between registers and memory is done by two instructions: `load` and `store`. The change framework could provide information on the usage of allocated data. For instance if always only some bytes of the allocated data are load/stored one could infer that the allocated memory is too big.

The change framework offers a broad range of analysis possibilities. Experienced engineers can write their own providers and thus are able to greatly enhance the analysis framework and provide custom information to change scripts.

3.3 Summary

The objective of this chapter was to give the reader a motivation as well as some real world scenarios for using the change framework to gain information about programs at run-time. First an example code fragment was presented. The change framework was used to answer some questions about the behavior of the code at run-time. This was done by using change providers in change scripts which were used to transform the application intermediate representation at run-time.

Chapter 4

The Low Level Virtual Machine

4.1 Overview

This chapter aims at introducing the Low Level Virtual Machine (LLVM) and the LLVM Architecture (LLVA). As already mentioned in Chapter 2, LLVM builds the infrastructure for the change framework. First the history behind the LLVM project is presented. The next section focuses on system concepts of computer architecture and discusses important concepts such as instruction set architectures (ISAs), application binary interface (ABI), or application programming interface (API). The following section then introduces the virtual machine concept with a focus on process virtual machines. Eventually the last section of this chapter gives a detailed overview of LLVA.

4.2 The History behind LLVM

The origins of the LLVM root in project called "The Lifelong Code Optimization Project" (LCO-Project) at the department of computer science at the university of Illinois at Urbana-Champaign. This project was led by Vikram Adve and had the following goal:

> "To enable modern application code to be optimized at link-time with all static library code, reoptimized at run-time with runtime information about system architecture and code behavior; and re-optimized offline in the field (i.e., between runs) using profile information gathered from production runs. We collectively refer to the latter two as post-link re-optimization.. . . " [Lif06]

To achieve this broad effort a new object code representation and compiler infrastructure for link-time and post-link optimization was needed. The LLVM provided this infrastructure. Within the project it is/was used for the following research efforts:

- Macroscopic data structure analysis and transformations
- Runtime optimization of native code
- Static analysis for software security and reliability
- Virtual processor architectures

The LLVM project grew out of this project and is now a top level project on its own. Nonetheless many of its key features root in the above requirements.

4.3 Computer Architecture

Before describing the low level virtual machine in more detail, it is important to define the concept of a machine. This section introduces principles of computer architecture in terms of machine, machine language, and its important interfaces. Complex systems, like computer systems, typically follow a layered approach. Every layer has its specific task and the interaction between layers is performed through contracts and interfaces. These contracts and interfaces have to be specified in an architecture. Furthermore the architecture outlines the overall design and concepts behind a computer system.

4.3.1 Computing Machines and Machine Language

Computers are also called computing machines, or just machines. The principal job of every computer is processing information. The core of a computing machine is often referred to as the processor(s), or the central processing unit(s) (CPUs). The CPU is generic in that sense that it takes a set of instructions as input and processes information based on these instructions. A computer program is a set of instructions encoded in a binary way that the computer can execute. This encoding is called "machine code" or just code. It is said that the program has to be in machine language to be executable. Thus machine language is a binary computer specific representation of a program.

Most processors consist of general purpose units: The arithmetical and logical unit (ALU), the control circuitry (or control unit), the memory, and the input and output devices (I/O unit). In the stored program or von Neumann architecture both instructions and data are represented within the computer's memory as binary code. This architecture is common today.

4.3.2 Programs and Machine Language

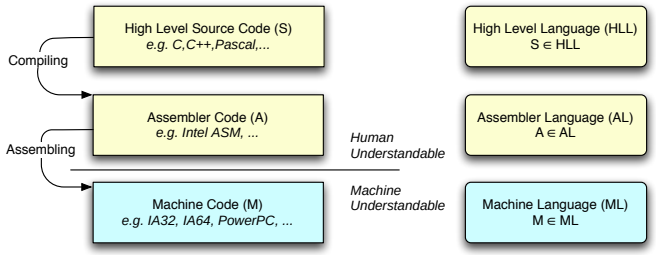


Figure 4.1: Translation from High-level Code to Machine Language

The processor or machine understands only one language, the machine language. In machine language the high level language constructs are represented by binary encoded processor instructions. A program in machine language representation is said to be executable on that machine. Every high level source code has to be translated to machine code to be executable by the processor. Figure 4.1 shows the typical translation from HLL code to ML.

4.3.3 Program Machine Interfaces

This subsection defines the interfaces between machines (either virtual or real) and application software. The interfaces represent the architecture. Interfaces have a big impact on the design of application software. Managing todays computer systems is a complex tasks. Adaptability, failure isolation, fault tolerance, and security are among the issues a computer system has to deal with. In the early days of computing, every system was controlled by a single program that was in control of the whole computer.

Interfaces are an important part of the architecture. At various layers the following are the most important interfaces:

- The instruction set architecture (ISA). ISA is used to refer to the characteristics of a processor with regard to its instruction set. The ISA forms the foundation for binary software compatiblity. It dates back to the development of the IBM 360 family in the early 1960s.
- The application binary interface (ABI). The ABI defines the interface to the machine from the application software point of view. For applications the OS is part of the machine.
- The application interface (API). The API refres to intefaces inside a high level language application. The API enables applications written to the API to be ported easily (via recompilation) to any system that provides the same API. The API tries to hide the underlying interfaces (such as the ABI, and the ISA) from the programmer.

4.4 The Virtual Machine Concept

4.4.1 Overview

The concept of a virtual machine is closely related to the machine concept. The machine abstraction is perspective dependent. From the application process's point of view the operating system is part of the machine. From the processor's (system) point of view the operating system is part of the executing system software. Because of this the term virtual machine itself is perspective dependent too. [SN05] characterize two types of virtual machines:

- Process virtual machines: Virtualizing a process environment.
- System virtual machines: Virtualizing the whole system environment.

This means that the type of virtualization depends on the type of machine to be virtualized. According to [SN05] virtualization consists of:

- The mapping of virtual resources or state, e.g., registers, memory, or files, to real resources on the underlying machine.
- The use of real machine instructions and/or system calls to carry out the actions specified by virtual machine instructions and/or system calls.

4.4.2 Process Virtual Machines

Computer programs are typically written in some programming language that are then compiled and linked as executable binaries that conform to a specific application binary interface (ABI). This so called user ABI is both processor (hardware) and operating system specific. Process level VMs provide user applications with a virtual ABI execution environment. There are different scenarios where this kind of virtualization is used. For instance such virtual execution environments can provide:

- Replication (multiprogramming)
- Emulation
- Portability and enhanced platform independence
- Optimization

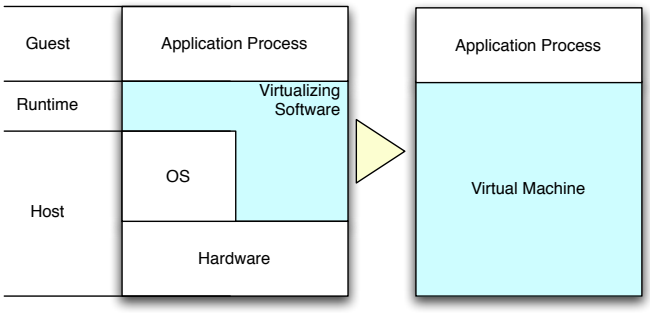


Figure 4.2: Process VM: The Virtual Machine runs as Application Process

As can be seen in Figure 4.2 the virtualizing software runs as an application process in the system environment. On top of the virtualizing software operates the real application software, which only sees the virtualizing software as its execution environment. The application that runs inside the virtual machine is called the guest. An ordinary application process is called a host process. Every guest application is seen as a host application by the operating system. The application itself does only know the guest machine. The guest machine is also called the virtual machine.

Guest applications are not compiled and linked towards the host process ABI, but use the guest process ABI, consisting of a guest user instruction set architecture (user ISA). Virtual machine ISAs are often called Virtual ISAs or V-ISAs. The next subsections list some common types of process virtual machines.

4.4.2.1 Multiprogramming

Most operating systems in use today already employ the concept of a process virtual machine. This machine is a same ABI and ISA execution environment. In a multiprogramming operating system multiple processes run isolated from each other simultaneously on the same operating system.

The supervisor system provides the application program with the illusion of being the only software program executing in the whole machine. This is done by providing each running process with its own virtual memory address space, which is mapped by the virtual memory subsystem of the operating system to physical memory. In addition, every process has its own execution context. Process execution is scheduled by the operating system scheduler. This is often referred to as preemptive multitasking. The scheduler runs every process for a given time and then interrupts the process to schedule the execution of another process.

The execution context for every process consists of

- Program counter (PC)
- Stack pointer (SP)
- Generic purpose registers
- Floating point registers
- Processor control registers (CPU state)
- Memory management register (virtual memory)

[SN05] call this a replicated process virtual machine that exists for all concurrently executing applications. In addition to replication, multiprogramming provides the user with failure protection. A defective application is less likely to render the whole system useless. Instead the user has the possibility to stop the faulty application while still using all other applications.

4.4.2.2 High Level Language Virtual Machines

Multiprogramming is a same ISA and ABI virtual machine (VM). This means the application is compiled for the host application and operating system, but instead of having one application in control of the whole machine, every application process gets a *virtual machine* that is managed by the operating system. This makes it possible to work with multiple applications in one real machine almost simultaneously.

So called high level language virtual machines (HLL VMs) exists for different reasons. One big issue is cross-platform portability. Not only that there exists multiple different hardware architectures (little vs. big endian, RISC vs. CISC, 32bit vs. 64bit), additionally every user application depends on the ABI and API. This means that even if a program is run on the same hardware architecture it has to be used on an operating system that uses the same ABI as the one which it was compiled for.

This tight coupling of application software towards their system platform is often unwanted and unnecessary. An approach using process virtual machines is one way to overcome this problem. So called HLL VMs are used to enable applications to be compiled against a high level language ISA and ABI. Only the HLL virtualization layer has to be ported to anothter specific target in order to load the guest application without recompilation on that target.

Cross-platform portability is not the only reason for the existence of HLL VMs. Another interesting aspect of HLL VMs is that the instruction set and the atomic operations can be augmented to better suite the needs of application programs written in high level

languages. This reduces the complexity of the application representation and makes it possible to understand binary programs easier.

The V-ISA of HLL VMs is often called bytecode or intermediate language. These ISAs are not suitable for direct hardware execution. Very popular modern HLL VM architectures are the Microsoft Common Language Infrastructure (MS CLI) as well as the Java Virtual Machine (JVM) Specification from Sun Microsystems. These two versatile platforms augment the ISA by:

- Object oriented features
- A memory model suitable for multi threaded applications
- Exception handling
- Access control and encapsulation
- Type safety
- Rich metadata support
- ...

Additionally these virtualization layers implement features like automatic memory management (garbage collection), reflective programming.

Instead of a simple Load/Store architecture, where values can either reside in memory or in registers, the Java VM defines the following locations where a value can reside:

- In an instance field: A field belonging to an instance
- In a static field: A field belonging to a class
- In an array
- In a local variable
- As a constant value

On the ISA level there exist the concepts of object classes, object instances, arrays, and constant pools. Additionally, a storage locations is typed. For instance, it is not possible to store a value in a storage location whose type is not assignment compatible with the value's type. This means that a type systems is part of the ISA.

Both the CLI and the JVM do not operate on values in registers but use a so called operand stack. A value has to be put on the operand stack to be used by instructions. An instruction is processed by taking its used values off the stack and pushing the outcome again on the top of the stack. After the instruction is processed the result is on the top of the operand stack and can be there transferred to the storage locations.

Many different process virtual machines exist with varying purpose. The main goal of this section was to introduce the concept of process virtual machines as well as show some motivations why process virtual machines exists. [SN05] provides additional information about process and system virtual machines.

4.5 The Low Level Virtual Machine Architecture

4.5.1 Overview

This section will look at the LLVM architecture. First the architecture will be classified. Then the ISA of the LLVM will be presented in more detail. Furthermore various representations of the LLVM ISA will be shown.

4.5.2 Classifying the LLVM Architecture

In Section 4.4 some fundamental virtual machine concepts where presented. As stated in Section 4.2 the LLVM projected originated in the LCO project with the objectives for lifelong code optimization.

The LLVM project's primary goal is to provide a compiler infrastructure. A compiler infrastructure is more than a compiler. A compiler is a program or a set of programs that takes a computer program in a source language, e.g. C, C#, Java, ..., and translates it into another language.

The LLVM project defines a complete virtual machine as a common model of inter-operation. The LLVM-IR builds a complete V-ISA, in a way, that a program can be compiled to LLVM object code and run by a virtualization layer, as in Figure 4.2.

In that the taxonomy of Section 4.4 the LLVM resembles a process VM. One of the key differences between HLL VMs and the LLVM architecture is the low level character. Programs compiled to LLVM object code do not require runtime support and lack high-level language features on the V-ISA level. This makes it possible to easily translate LLVM object code to target machine code in a traditional ahead-of-time (AOT) compilation model.

4.5.3 High-Level Type Information, Low-Level Intermediate Language

In this subsection we will define the meaning of *high-level* and *low-level* in the context of type information and intermediate languages. LLVM by definition is a so called low-level virtual machine. LLVM, compared to other process virtual machines characterizes itself as having a low-level architecture.

Virtual machines, as real machines, are characterized by their ISA. A low-level virtual machine therefore has a low-level ISA. A low-level ISA operates on a low-level intermediate representation (IR). The classification in this section is based on [Muc98]. Figure 4.3 shows different types of intermediate languages (ILs) and differentiate between:

- High-level intermediate languages: Are representation that are very close to the original source code. One examples are abstract syntax trees (ASTs).
- Medium-level intermediate languages: Are already closer to a low-level representation. This means they lose some characteristics of the source code representation.

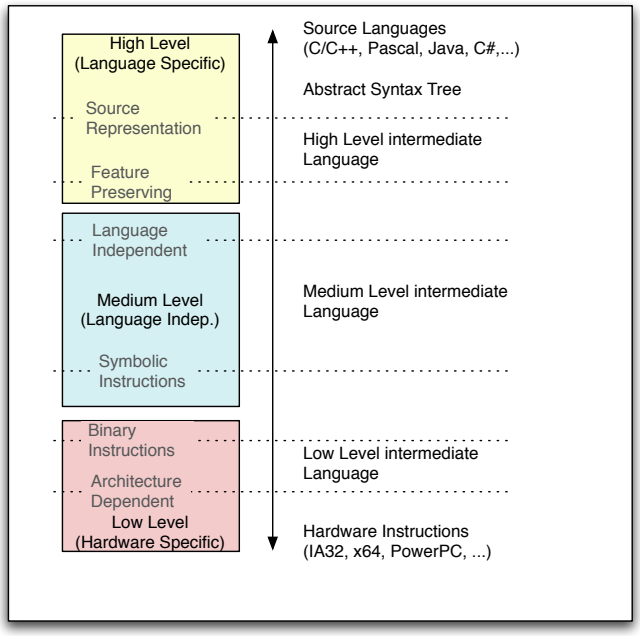


Figure 4.3: Different Program Representations

- Low-level intermediate languages: These ILs are more target specific. A typical compiler uses a low-level representation to generate target code. The goal of lower-level ILs is to mimic the various target architectures as much as possible, at the same time retain abstract enough that low level tasks can be performed target independent, if possible.
- Multi-level intermediate languages: These ILs include features that are best viewed as representing multiple levels combined in one IL.

The LLVM intermediate representation is a multi-level IL. On the one hand symbolic information is lowered to offsets, on the other hand LLVM supports high level type information and supports first class support for functions and function types.

4.5.3.1 High-Level Type Information

LLVM preservers high-level types. In LLVM IR types are not lowered to being either integral or floating points. The instructions are typed. This means that the LLVM intermediate representation has high-level type information in all its instructions. This enables much more powerful program analysis and data-flow analysis.

4.5.4 LLVM Virtual Instruction Set Overview

The intermediate representation of the LLVM compiler infrastructure is a typed virtual instruction set (V-ISA). [ALB+06] define the major design requirements of the LLVM V-ISA as:

- Driven by the needs of compiler technology.
- Universal enough to support arbitrary user and system software.

[Lat02] characterizes the LLVM instruction set architecture as

> "... The LLVM instruction set represents a virtual architecture that captures the key operations of ordinary processors but avoids machine specific constraints such as physical registers, pipelines, low-level calling conventions, or traps. LLVM provides an infinite set of typed virtual registers which can hold values of primitive types (integral, floating point, or pointer values). The virtual registers are in Static Single Assignment (SSA) form ..."

The LLVM instruction set can be defined as having

- a three-address code architecture in SSA Form,
- high-level type information preserved in the V-ISA,
- a load and store architecture using typed pointers,
- type-safe pointer arithmetic,
- support for distinguishing safe and unsafe code via a cast instruction,
- support for explicit memory allocation and a unified memory model,
- support for function calls and exception handling,
- a graph-based in memory representation, represented via a set of libraries,
- support for textual assembler representation, and
- support for a binary virtual object code representation.

4.5.5 Three-address Code Architecture in SSA Form

LLVM IR as opposed to many HLL VM IRs is not based on an operand stack. Instead LLVM is a load and store architecture with an orthogonal three-address instruction set format.

This resembles machine code very closely. LLVM defines the following instruction classes:

- Control-flow instructions (also called terminator instructions, these terminate basic blocks)
- Binary operators
- Logical operators
- Binary comparison operators
- Memory operators
- Other operators
- Meta instructions

Meta instructions (also called pseudo instructions) are no instructions on their own. These instructions encode information into other instructions, for example `volatile` can be used in conjunction with a `store` to form a `volatile store`. The arithmetical and logical operations, in three-address form are `add`, `sub`, `mul`, `div`, `rem`, `not`, `and`, `or`, `xor`, `shl`, `shr`, and `setcc`. `setcc` is a set of comparison instructions with different operators:

$x == y$	`seteq %x, %y`
$x != y$	`setne %x, %y`
$x >= y$	`setle %x, %y`
$x <= y$	`setge %x, %y`
$x > y$	`setlt %x, %y`
$x < y$	`setgt %x, %y`

Three-address form defines the tuple: $< operation, in1, in2, out >$.

4.5.5.1 Typed Polymorphic Instructions

Every value in the LLVM V-ISA is typed. There is one definition of an add instruction, called `add`. This instruction can operate on several different types of operands, which reduces the number of opcodes needed. The types of the operands of an instruction define its result type and the semantics of the operation. Type rules have to be followed strictly and are defined in [LLV06c].

4.5.5.2 Explicit Control Flow Information

Control flow information in LLVM is explicitly encoded in the form of a control flow graph. A control flow graph (CFG) is a directed graph that provides information about the flow of control in a set of instructions. A CFG is a directed graph G, defined as:

$G := (N, E)$
$N := \{n | \text{n is a basic block}\}$
$E := \{(n_i, n_j) | (n_i \in N) \wedge (n_j \in N) \wedge \text{possible transfer of control from block } n_i \text{ to } n_j\}$

Where a basic block is a set of instructions which is always executed together. Control always enters a basic block at its first instruction and leaves the block at its last.

```
while ( i < 10 ) {
  if ( i % 2 == 0) {
    puts("even");
  } else {
    puts("odd");
  }
  i++;
}
puts("end of loop");
```

Listing 4.1: Loop Code Fragment (C language)

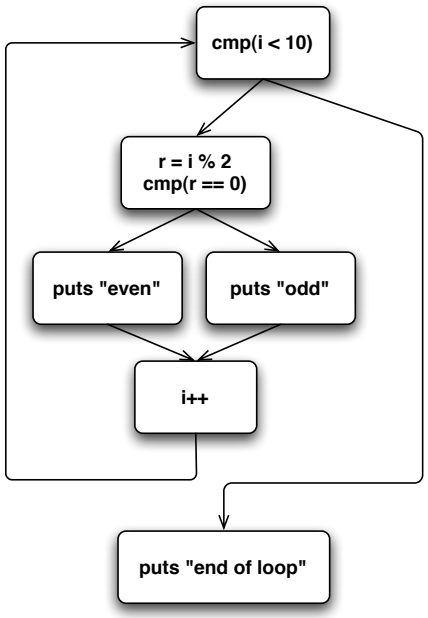


Figure 4.4: A Simple CFG

Figure 4.4 shows a simplified *CFG* for the code fragment of Listing 4.1. Nodes in the figure represent basic blocks, edges the control flow. Such a *CFG* provides all the possible run-time control-flow paths. Listing 4.1 shows a simple loop and a conditional branch inside the loop body. As can be seen in the figure the loop creates a cycle in the execution path. The last node in the loop body gives control back at the loop header. The loop header decides when to exit the loop. Eventually control is transferred to the node in the bottom of the figure.

The LLVM V-ISA has direct support for basic blocks and control flow instructions. The graph-based in-memory representation is based on a *CFG*. A function consists of a set of basic blocks and each basic block consists of a set of instructions. Every basic block has exactly one terminating instruction. This instruction is a control transferring instruction. In Subsection 4.5.6 we will discuss this in more detail.

4.5.5.3 Static Single Assignment Form

LLVM instructions operate on registers. Instead of having a limited number of registers, as hardware ISAs have, LLVM has an infinite number of registers. Within a code fragment, one register can only be defined once. This means that for every new definition a new register has to be used. If a program is in Static Single Assignment (SSA) form [CFR+91], every variable is defined exactly once. Furthermore each use of a variable refers to a single definition and the use is dominated by the definition of the variable. The definition always (no matter via which control-flow path) precedes the use of a variable.

LLVM registers are in SSA form. Whenever control flow merges a definition of a variable might depend on the execution path. This means there could be several definitions of

the variable, depending on the run-time path taken. To overcome this situation, the SSA form uses a ϕ function. Such a ϕ function is of the form:

$$x := \phi((a, BB_a), (b, BB_b), (c, BB_c), (d, BB_d), \ldots)$$

ϕ takes several tuples. Each tuple consists of a value and the basic block, where the execution came from. The ϕ function then selects the value of the matching execution path. ϕ functions are the first entries in a control flow merging basic block. For every incoming basic block, the ϕ function has a value. LLVM uses a ϕ instruction that provides exactly this functionality. In the following example, we show the use of a ϕ instruction in LLVM code.

```
<result> = phi <type> [<val0>,<label0>], ...,
```

The three listings below show a simple function, `loopedSum`, which calculates the sum from 1 to N by using a loop. The Listing 4.2 shows the ordinary version of the function in plain C. A `for` loop is used to iterate over an induction variable i. The result is stored in the variable result, which is increased incrementally.

```
int loopedSum(int N)
{
  int result = 0;
  for (int i = 1; i <= N; i++)
  {
    result = result + i;
  }
  return result;
}
```

Listing 4.2: loopedSum in C

Listing 4.3 shows the same code algorithm, but in SSA form, using named blocks and C style gotos. Except the ϕ function this code is still plain C. One can see that in the LoopHeader the variables i, and result, are defiend using the ϕ node. The LoopHeader has 2 incoming blocks, named Entry, and Loop. Thus phi gets two arguments, one if control is reached via the Entry block, the other if control reaches LoopHeader via the Loop block.

```
int loopedSum(int N)
{
Entry:
  register int i0 = 1;
  register int result0 = 0;
LoopHeader:
  register int i1      = φ(<i0,Entry>,<i2,Loop>);
  register int result1 = φ(<result0,Entry>,<result2,Loop>);
  if ( i1 <= N )
     goto Loop;
  else
     goto LoopExit;
Loop:
  register int result2 = result1 + i1;
  register int i2 = i1 + 1;
  goto LoopHeader;
LoopExit:
  return result1;
```

```
}
```

Listing 4.3: loopedSum in Pseudo C Code in SSA Form

Finally Listing 4.4 shows the function in LLVM assembly code. This is very similar to Listing 4.3, except that LLVM instructions are used instead of C language expressions and statements. Since names only apply to values, constants cannot be named. This is why the basic block `Entry` in Listing 4.4 differs from the one in 4.3. In LLVM the initial values are introduced as operands of the `phi` instruction.

```
int %loopedSum(int %N)
{
Entry:
  br label %LoopHeader
LoopHeader:
  %i1 = phi int [1, %Entry], [%i2, %Loop]
  %result1 = phi int [0, %Entry], [%result2, %Loop]
  %cond = setle int %i1, %N
  br bool %cond, label %Loop, label %LoopExit
Loop:
  %result2 = add int %result1, %i1
  %i2 = add int %i1, 1
  br label %LoopHeader
LoopExit:
  ret int  %result1
}
```

Listing 4.4: LLVM Assembly Code of the loopedSum

4.5.5.4 Type Information

The LLVM V-ISA is a strictly typed representation. Every value in LLVM has to have a type. Values can be LLVM registers in SSA form or memory locations. This type information provides more global information about the application. Furthermore the type information can be used to check the type safety and to check for errors in the LLVM code.

The LLVM type system consists of C-like types:

- Primitive types
- Complex types

The primitive types are:

- void
- bool
- signed and unsigned integers from 8 to 64 bits
- single precision (32 bit) and double precision (64 bit) floating point types (float, double)
- the opaque type

The complex types consist of:

- pointers
- arrays
- structures
- functions

The LLVM type system tries to be language agnostic. No direct representative of higher level types, for instance classes, are used. These constructs have to be mapped on LLVM types. As already mentioned all LLVM instructions are strictly typed. Additionally LLVM instructions have restrictions on the types of their operands. The `add` instruction requires its operands to be of the same type, which has to be an arithmetic type (i.e. integral, floating point). The type of the result is the type of its operands.

```
struct MyStruct {
  int x;
  float *pointerToY;
  int array[10];
};

int main(int argc, char** argv);
```

Listing 4.5: Some C Types

```
%MyStruct = { int, float*, [10 x int] }
%main = int (int, sbyte**)
```

Listing 4.6: Equivalent LLVM Types

Listing 4.5 shows a C structure and a function declaration. Listing 4.6 shows the equivalent types in LLVM assembly code.

4.5.5.5 LLVM Memory Model

LLVM programs consist of a stack and a heap. The LLVM V-ISA has explicit typed memory allocation instructions. The memory allocation and free instructions are:

- `malloc`
- `alloca`
- `free`

These instructions closely resemble the C runtime functions for memory allocation and reclamation. The `malloc` instruction allocates one or more elements of a specific type on the heap, returning a typed pointer of the element's type which points to the new memory.

The `free` instruction releases the memory allocated by the `malloc` instruction. The `alloca` instruction resembles the `malloc` instruction, but allocates the objects on the current stack frame instead of on the heap. Stack storage is automatically managed, which means that there is no need for a `free` instruction for stack allocated objects.

Furthermore according to [Lat02] all addressable objects in LLVM are explicitly allocated:

- Stack allocated locals: Are allocated using the `alloca` instruction.
- Dynamically allocated memory: Are allocated using the `malloc` instruction
- Global values: These are global variables and functions.

Global values declare regions of statically allocated memory and are accessed through the value of the object, e.g. the name of the global variable refers to the address. Memory objects are always accessed by their address. Furthermore LLVM does not have an address-of operator. Every variable referring to a memory object is always a pointer to that value. All memory traffic occurs in LLVM via `load` and `store` instructions.

4.5.5.6 Function Calls and Exception Handling

The LLVM V-ISA has first class support for functions and exception handling. Depending on the exception handling needs, functions can be called using either `call` or `invoke`. In any case the argument passing is done atomically by these instructions. Parameter passing rules are not part of the generated code, but are abstracted by the VM. This distinguishes the LLVM ISA from machine code ISAs. In machine code ISAs, like for instance the IA32, the compiler has to deal with the parameter passing, which can be done using registers and/or using stack slots. Furthermore LLVM has an infinite set of registers in SSA form so there is no need for caller/callee saved registers. These facts make it much easier to work with LLVM IR than to work with target machine code.

The `call` and `invoke` instructions take a pointer to a function as well as the parameters to pass. As in C all parameters are passed by value. LLVM implements stack exception handing in a way that the presence of exception handling causes no extra instructions to be executed when no exceptions are thrown. If an exception is thrown the stack is unwound, stepping through the return address of function calls on the stack. LLVM holds a static map of return addresses to exception handler blocks that are used to invoke handlers during unwinding.

The difference between the `call` and the `invoke` instruction is that the `invoke` instruction terminates the basic block. The call site explicitly declares to handle an exception. Hence if the caller throws an exception, control flow might continue at the exception handler not right after the `invoke` instruction. When using the `call` instruction the `call` site will never be notified if the callee threw an exception.

```
// function call ignoring possible exceptions
func();

// function call, handling exceptions
try {
  func();
} catch( ... ) {
  puts("exception during func call!");
}

puts("after func call");
```

Listing 4.7: C++ Call Site, Showing Function Call and Exception Handling

Listing 4.7 shows a simple C++ code fragment. First a function is invoked without handling exceptions. After that the same function is called. This time the function is surrounded by an exception handler.

```
    ; the first invocation of does not have an exception
    ; handler. if an exception is raised, the call site
    ; is leaved.
    call void %func()

    ; calling func with exception handlers installed
    ;
    invoke void %func() to label %OkLabel except label %CatchLabel
  OkLabel:
    ; ... no exception occured; no branch
    ; ...
    call int (sbyte*)* %puts(sbyte* "after func call");
    ; ...
    ; return
  CatchLabel:
    ; ... exception occured
    call int (sbyte*)* %puts(sbyte* "exception during func call!");
    br %OkLabel
```

Listing 4.8: LLVM Assembly Code, Showing invoke and call

Listing 4.8 shows the LLVM version of the C++ code fragment. The first function call is performed using the `call` instruction. The second function call is done using the invoke instructions. The `invoke` instruction takes two additional parameters, the label if no exception is raised and a second label in the exception case.

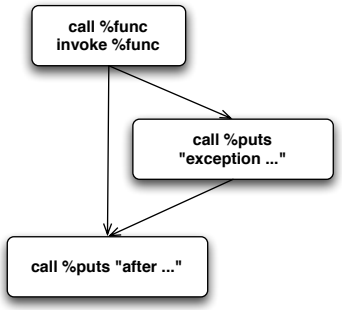


Figure 4.5: LLVM `call` and `invoke` CFG

The `call` instruction does not terminate a basic block, but the `invoke` instruction does. Figure 4.5 shows the simplified CFG of Listing 4.8. As one can see the invoke introduces two control-flow paths. One in the case of normal execution and one in the case of an exception.

4.5.6 Graph-Based In-Memory Representation

LLVM defines three different representation formats: two serialized formats and one in-memory format. The two serialized formats consist of one textual format and a

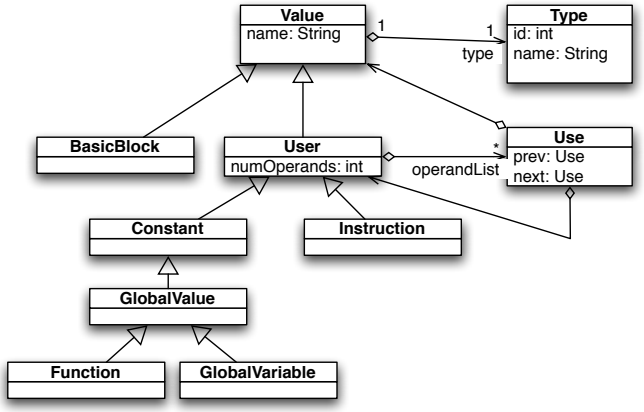


Figure 4.6: LLVM Value-User Type Hierarchy

binary format suitable for persistent storage of LLVM object code. The in-memory representation is graph-based. This means that the memory representation resembles a direct acyclic graph (DAG). The serialized representations are used as an interface between LLVM tools as well as to write and read LLVM assembly code. The in-memory IR is implemented as a C++ class library. [CP95] describes the advantages of a graph-based intermediate representation. Figure 4.6 shows the most generic classes in an Unified Modeling Language (UML) class diagram.

The `Value` class represents all values in an LLVM code. Every value has a type. Furthermore the LLVM IR object model distinguishes between two primary values, instructions, and constants. Every instruction is an instance of class `Instruction`. Every constant an instance of `Constant`. Global addressable values are represented by the `GlobalValue` class. Representatives of global addressable values are global variables and functions. Furthermore the `Instruction` and `Constant` types are `User` types. As can bee seen a user has a reference to one or more uses. A use is an instance of class `Use`. Such a use is a holder that provides access to both the used value and the user of that value. This abstraction eases use analysis. One only needs a list of uses in order to access both the users and used values.

Figure 4.7 shows the UML class diagram of the instruction hierarchy. As we can see there are many different instructions. The derived classes depend on details of the instruction they model. For instance there is a class `UnaryInstruction` which represents all instructions that only take one parameter. A special class of instructions are the instances of `TerminatorInst`. All of them terminate a basic block, thus are always the last instruction in a basic block. `TerminatorInst` are also called control flow instructions. Currently they are:

- `ReturnInst`
- `BranchInst`
- `SwitchInst`
- `UnreachableInst`
- `InvokeInst`
- `UnwindInst`

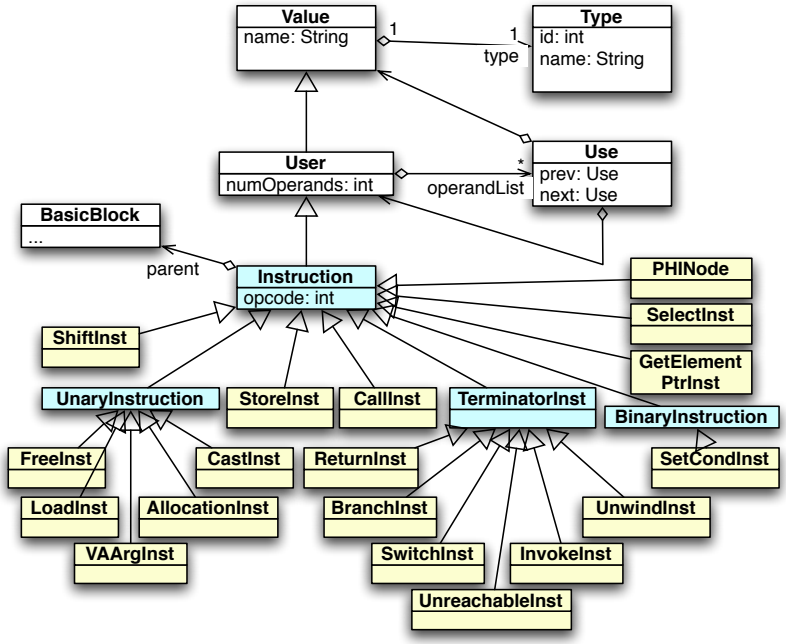


Figure 4.7: LLVM Instruction Hierarchy

These instructions provide the pointer to the next basic blocks. Instructions of type `TerminatorInst` are needed to represent the CFG within LLVM bytecodes. Additionally since every instruction is a user and therefore also a value, instructions can reference other instructions directly. This implies that the in memory representation has no direct concept of a register. Everything that is a value is a register or constant value. Instructions can be named thus providing a way to create named registers. These named registers are then used when the graph-based representation is serialized.

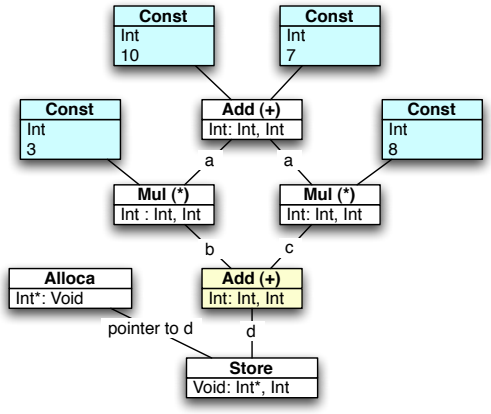


Figure 4.8: LLVM In Memory Representation

```
register int a = 10 + 7;
register int b = a * 3;
register int c = a * 8;
int d = b + c;
```

The above C code fragement stores a complex expression in the variable d on the current stack frame. The variables a, b, c, are not stored on the stack they are kept in registers. Figure 4.8 shows the expression as a graph of values. Every box represents an instance of `Value`. The first element is the type in textual representation. Two values have the same type if their type identifier is lexical equivalent. In the in memory graph, types are represented by pointers to shared instances of class `Type`.

4.5.7 Bytecode - Binary Virtual Object Code Representation

The LLVM architecture is very closely related to compiler architecture. A compelling feature of the LLVM architecture is that it has a binary object code representation. A program can be compiled to a binary LLVM object code. This object code can be later compiled to machine code or can be loaded by a dynamic execution engine that generates machine code by interpretation or binary translation.

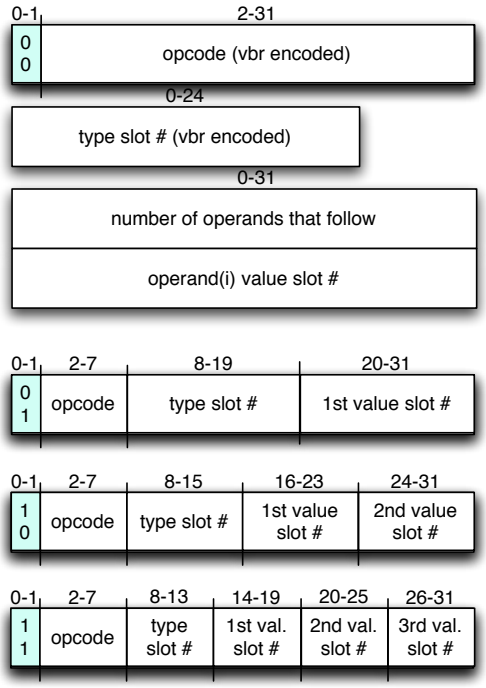


Figure 4.9: LLVM Binary Instruction Format

Figure 4.9 shows the instruction encoding of LLVM instructions. A binary LLVM code is represented by a module. A module can have multiple functions and global variables. The formal layout of the LLVM bytecode is defined in [LLV06a]. All values are encoded in variable bit rate (VBR) encoding. In this encoding the most frequently used values take the lowest numbers. This makes the bytecode more compact. The discussion of the LLVM bytecode format is limited to the instruction encoding. As can be seen in Figure 4.9 there are 4 types of instruction formats. The least significant two bits determine the format of the instruction.

- `00` - Variable number of operands

- `01` - Instructions taking a single operand
- `10` - Instructions taking two operands
- `11` - Instructions taking three operands

The encoding guarantees that in most cases an instruction only takes 32 bits. In order to achieve that, the values and types are not referenced directly, but through a type and value table.

4.5.8 Summary of the LLVA

This introduced the LLVA and the most important aspects of its V-ISA as well as the different representations and a comparison to other ISAs. The LLVA is a mid-level IL, it resembles a low-level representation with high-level type information. LLVA is in SSA form and has in-memory as well as serialized representations.

4.6 Summary

This chapter introduced the LLVM and the LLVA. The LLVM project originated at the university of Illinois at Urbana-Champaign. The LLVM is a process virtual machine. It is a low-level virtual machine with high-level type information. LLVM provides an infinite set of typed virtual registers which can hold values of primitive types. Furthermore the virtual registers are in Static Single Assignment (SSA) form. Additionally to the in-memory representation the LLVM also has two serialized representations, one textual, and a binary representation. Part of the LLVM project is a back-end for the GNU Compiler Collection (GCC) project. This enables existing C/C++ applications to be targeted to LLVM.

Chapter 5

The Change Framework

5.1 Overview

The change framework is a system for dynamic program instrumentation and analysis based on the Low Level Virtual Machine (LLVM) architecture. Chapter 4 covers the underlying LLVM architecture. This chapter describes the instrumentation and analysis architecture in greater detail. Starting from an architectural perspective, we will discuss the individual parts and concepts of the framework. The main objective of the change framework's design is to be able to dynamically (at run-time) instrument applications. As discussed in Chapter 2 the framework uses a technique called bytecode instrumentation. The change framework extends the LLVM execution engine. Hence the intermediate representation that gets transformed is the LLVM graph-based in-memory intermediate language as described in the last chapter.

The LLVM dynamic compiler is used to binary translate parts of the bytecode to machine code on an as needed basis at run-time. Instrumentation of the application logic is performed by transforming the IR. Since the compiler is available at run-time, recompiling the changed IR is possible without stopping and restarting the application. This enables dynamic instrumentation. The design implies that the application's code has to be compiled to LLVM bytecode to be run within the change framework. Part of the LLVM project is a version of the GCC compiler suite that includes support for generating LLVM bytecode from C/C++ and ObjectiveC source files.

This way many existing C/C++ applications can be compiled to LLVM bytecodes and hence are able to run within the change framework. Especially low level system code can be instrumented too. Typically high level virtual machines like Java are lacking direct support for compiling low level, non object-oriented system code to their respective byte code. The project [Bri] even aimed at porting the entire Linux kernel to LLVA.

5.2 Change Framework Architecture

This section covers the high level architecture of the change framework. As can be seen in Figure 2.1 the system consists of:

- The application process
- A monitor process

The application under analysis is running inside the *application process*. As mentioned above the application is compiled to LLVM bytecode and loaded using the LLVM execution engine. After loading the bytecode, the in-memory intermediate representation (IR), as discussed in Chapter 4, is used to perform instrumentation. The IR is not interpreted. The execution of the application is performed in its native (machine code) form.

5.2.1 The Change Concept

The central concept that also gives the name to the change framework are *changes*. As can be seen in Figure 2.1 change scripts are written in the change language and represent one or more *change*. A *change* is the term that refers to an alteration to the application's IR. The language for writing changes will be discussed later in this chapter.

A change is applied to one or more points (*change points*) in the execution of the application under analysis. A change consists of a number of statements, that are inserted at these points. As can be seen in Figure 5.1 changes additionally contain so called *predicates*. Predicates are used to control which of the potential change points will be affected by this change. Predicates give the change author control where the change is to be applied. We will see later that *change providers* exposes potential change points and check predicates.

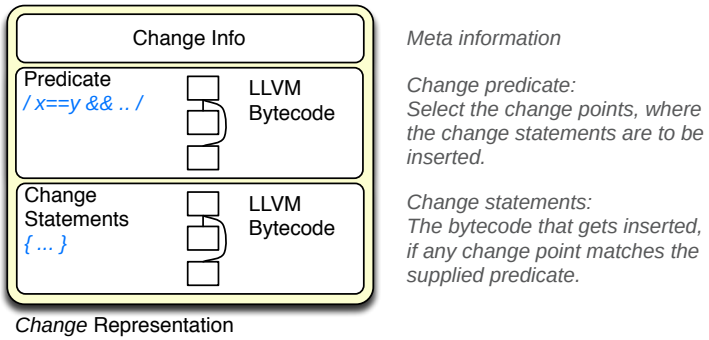


Figure 5.1: Components of a Change

To summarize, a *change* together with a *change provider* control the transformation process on the application's IR. Changes originate in the monitor process. They are written in the change language and compiled to LLVM bytecode. After that the monitor process sends changes to the application's process using the *change protocol*. The change protocol operates over a communication channel between the monitor and the application's process.

The two most important requests are:

- Requests to *apply* a certain change to the running application.
- Requests to *unapply* a change or a set of changes. This reverts the changes performed to the code. After reverting a change the application should work as if the change would never have been applied.

In addition the communication channel provides a way to communicate results from the application back to the monitor process.

5.2.2 Change Application Overview

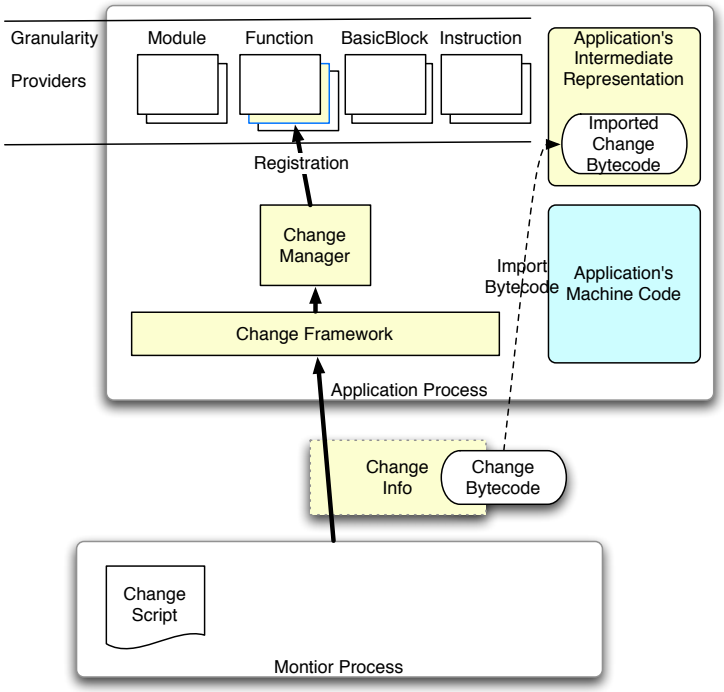


Figure 5.2: Applying a Change

The application's process listens for change requests. When a change request enters the application's process, it is analyzed and handed to a central component called the *change manager*. As can be seen in Figure 5.2. If the request is a change *application* request, then

- the change's bytecode is imported and added to the application's IR, and
- the change is registered with its corresponding *change provider*. After registering the change gets a unique registration id. This registration id is called a cookie. The cookie is returned to the monitor process and is used to refer to that change.

While a change can access information from many providers, a change has exactly one *transforming provider*. This provider is chosen by the change author and is part of the change identifier. That provider is the responsible component in the application process, we refer to this provider as the change's *responsible provider*. Figure 5.2 shows the actions performed when a change arrives: The change manager registers the change with its responsible provider.

The responsible provider has the task of checking which points in the execution of the application are to be affected by this provider. This is done by evaluating the change predicates. Further the provider also has to transform the application IR to incorporate

the change at matching *change points*. The change application process traverses the application's IR bytecode model. At every level (module, function, basicblock, instruction) providers are informed. The responsible provider then checks, whether

- there is a change registered, and
- the change predicate matches the current context.

If both conditions hold, the change is weaved into application's IR. For undoing the change later, a change log is kept. After the change is weaved into the application IR, the application IR is marked as dirty. The application code periodically checks whether the machine code should get recompiled. If it detects that the application IR got modified, the dynamic compiler is run. After recompiling and relinking the application runs the new, modified machine code.

5.2.3 Change Unapplication Overview

If the entering request is a change *unapplication* request, the framework performs the following tasks:

- The change manager gets informed, that a change with a certain change id (cookie) should be unapplied.
- The change manager checks the change for validity, especially if the change has really been registered and whether the change is already/still applied.
- A global undo log is used to remove the added instructions and global variables.
- Afore imported functions that belong to the change being unapplied can be removed. Memory can be freed.
- The change is removed from the internal bookkeeping structures. It cannot be unapplied more than once.

After the change removal process, the application's IR is marked dirty and the dynamic compiler will generate new machine code for the changed application parts. The effects of the change application process are undone. The change is not applied anymore.

5.3 Change Provider Architecture

Change providers form a crucial part in the system above. Change provider live in the application's process and handle the transformation of the IR as well as providing context information. The term provider stems from the fact that a provider encapsulates these services and allows the system to be extensible. One can add custom providers and register them with the system. Providers have to be implemented in C/C++ and directly access the LLVM Compiler API. After registration, a provider can be used by the change author.

Change providers operate on a predefined structural granularity. The granularity represents the bytecode level scope for which the provider is responsible. Currently these are

- per module,
- per function,
- per basic block, and
- per instruction.

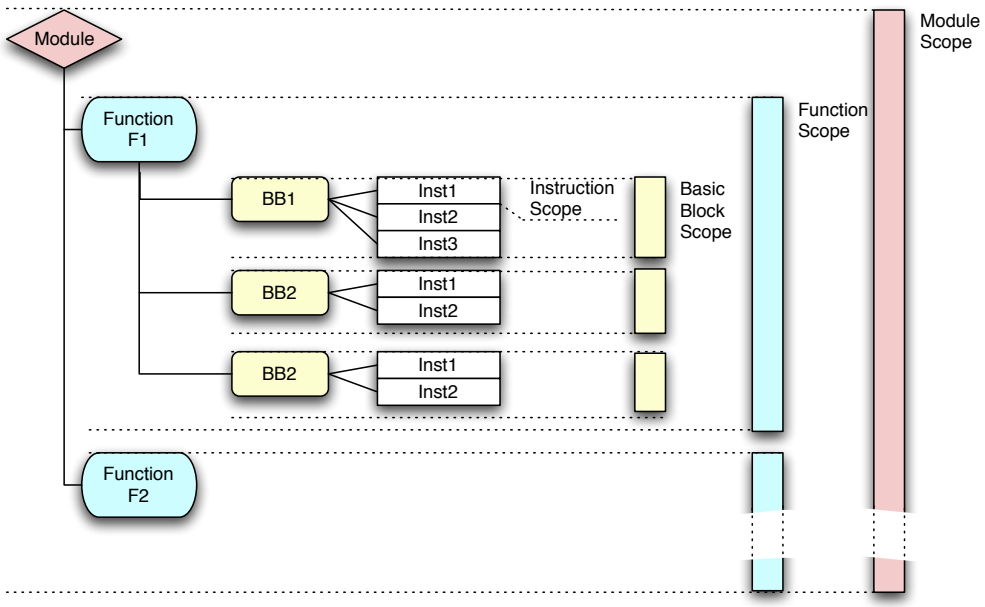


Figure 5.3: Bytecode Scopes

Given a certain granularity, every change provider *provides* potential *change points*, and static *context information*. The concept of nested scopes applies to context information. As can be seen in Figure 5.3, basic blocks are within the scope of a function, and instructions are inside the scope of each basic block. The context information of a function is valid throughout its scope.

A change's responsible provider always operates on a specific granularity and is scope-aware. Change statements and change predicates of a specific granularity can thus use static context information of its container objects. This way a basic block change predicate, for instance, can access context information of the container function and the container module. An instruction change has additionally access to the basic block that contains the instruction, and so on.

The main tasks of a provider are:

- Exposing potential change points.
- Evaluating change predicates to obtain change points.
- Updating information about its static context.
- Transforming the application's IR at change points.
- Keeping undo records.

As an example we take a provider that works on a per function granularity. Such a provider exposes potential change points at the beginning and at the end of a function.

The provider is free to filter these change points before hand. A provider with function granularity operates on a function at a time. Thus it only exposes points in the execution of the application that are on function level such as entering and leaving a function.

Our example function provider could expose information, such as:

- The name of a the function.
- The signature of the function.
- The number of arguments.
- Its return type.
- ...

All the information above is local to the function and known statically. Additionally to that information, such a per function provider could expose change points.

5.3.1 A Sample Application

Listing 5.1 shows a sample application. This application is written in the C language and has the following structure: It consists of one translation unit (here called a module), a global variable, called `names` which is an array of strings. Furthermore it has a function called `say`, one called `hello_world`, and another one called `main`. `main` is the module's *entry point*.

```
#include <stdio.h>

char *names[] = { "Hello", "World", "Application", NULL } ;

void say(const char * value)
{
  printf( "Say: %s\n", value);
}

int hello_world( ) {
    char **name_iter = names;
    while ( *name_iter ) {
      say( *name_iter );
      name_iter++;
    }
    return 0;
}

int main(int argc, char** argv) {
   return hello_world( );
}
```

Listing 5.1: Sample Application

The application's behavior is as follows: On start `main` is invoked. `main` in turn calls the `hello_world` function. `hello_world` simple prints all elements of the global array on the standard output by calling the `say` function. `say` takes a string and prints it to `stdout` using the C run-time function `printf`.

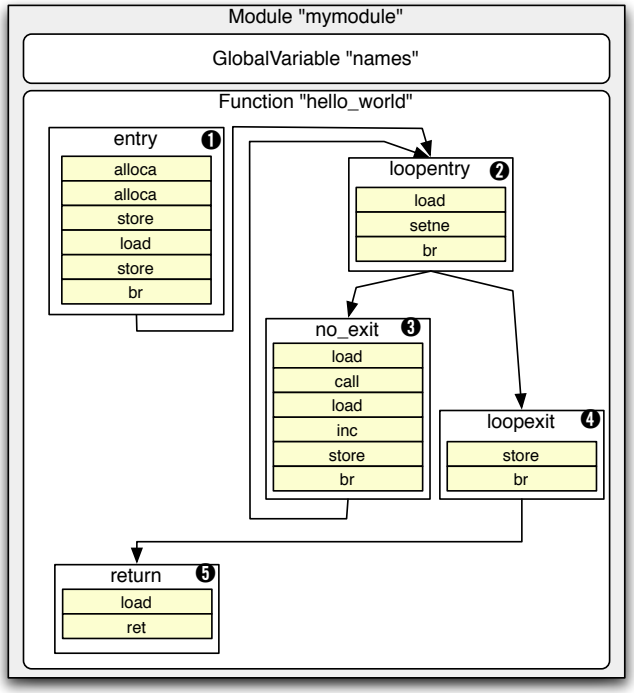


Figure 5.4: Sample Application LLVM IR for Hello World

Figure 5.4 shows the LLVM IR of the module and the `hello_world` function in a graph-based representation. This representation shows the module, the global variables, and the function `hello_world`. The function `hello_world` is represented as set of basic blocks, which are connected together by control flow instructions. Such a representation is also called a control flow graph (CFG). Please refer to Chapter 4 for more theoretical background. Each basic block is shown with its instructions. In favor of clarity Figure 5.4 only shows the opcodes of the instructions.

5.3.2 Change Points and Change Point Trajectories

Change points always represent events in the execution of the program. The collection of change points over a period of time is referred to as a change point trajectory. During the application of changes every change provider that has changes to apply checks whether change points match. This is done by evaluating the change's predicate with the current static context information. When a change point occurs, the change is weaved into the application's IR.

```
procedure AFunctionProvider_onFunctionEnter(provider:Provider, changes
    :Array of Change)
begin
  for each change in changes
     if change.predicate(provider.ctxInfo) then
        ; this is a change point
        insert change.changeStatements at this point in the model
     end
  end
end
```

```
...
```

Listing 5.2: Change Application - Changepoint

When a change is applied, the application's IR is traversed and at potential change points, the incoming predicate is matched. If the evaluation of the predicate yields true for a specific point in the structure of the application, the change is instrumented into the IR. Based on the way the application's static structure is traversed, a certain trajectory of change instrumentation is performed.

The pseudo code in Listing 5.2 shows the logic behind a function provider's change point called `onFunctionEnter`. The important point is that the change point exported as part of the provider definition, maps to logic inside the provider's implementation. This logic has to evaulate the predicate and instrument the change into the point, if needed.

Whether or not the application's IR is changed by a specific change is determined by the static context information provided by the change providers. Later when the execution of the new code begins, change points actually are executed. This is the dynamic trajectory of change points, which depends on the run-time execution paths taken by the application.

5.3.3 Provider Context Information

As stated in the last section one important job of every change provider is to export information about its static context. This static context is used to evaluate change predicates. Decision whether a point in the execution of the application is a change point is done by the change provider.

The information collected is subject to the provider implementer. Whatever the implementer chooses to collect is available to the change author. The information collected is purely static. No information about the run-time state of the application needs to be known. This is important as the transformation process has to be as efficient as possible.

```
function provider functionProvider {
  points  {OnEnter, OnLeave} ;

  int id;
  string functionName;
  int argCount;
  int isVarArg;
};
```

Listing 5.3: Change Provider Information

In Listing 5.3 one can see the context information exported by the provider `functionProvider`. This information matches the information the implementation of that provider is able to expose. By this means the provider is able to export information to the change framework. This information is read-only for the change author. It can be used both to decide whether to apply the change (e.g. inside a predicate) and inside the change statements.

5.3.4 Summary

In this section the change concept was introduced. Change providers are the interface between the change author and the execution engine. Change providers are written in C/C++ and work directly with the LLVM API to manipulate the intermediate representation. The provider architect has to choose a granularity that best suits the needs of the provider. Additionally the architect exports information, called static context information, to the change author.

When writing a change script the change author chooses one responsible provider. This provider is responsible for checking whether and when a change should be weaved into the application's IR. The next section formally defines the change language.

5.4 Change Language

5.4.1 Overview

The change language provides an efficient way to write changes. As can be seen in the next subsection, change language statements are syntactically a subset of C. The change language is a domain language for authoring changes. Change providers, change predicates, and change statements are first class language concepts. Some important properties of the change language are:

- The change language is used to create one or more change definitions.
- Available providers have to be declared inside the change language in order to be used.
- Each *change definition* consists of:
  - A responsible provider
  - A change predicate
  - A block of change statements

5.4.2 Syntax

The syntax of the change language is described as a grammar in Listing 5.4 in Extended Backus Naur Form (EBNF) [Wir77]. In this section we discuss syntactic properties of the change language based on this grammar.

```
TranslationUnit    = {ProviderDecl | ChangeDefinition}.

ProviderDecl       = ("module" | "function" | "basicblock" |
    "instruction") "provider" ident "{" [ProvChangePointDecl ";"] {
    ProvFieldDecl ";"} "}".

ProvChangePointDecl = "points" "{" ident {"," ident} "}"
ProvFieldDecl      = TypeSpecifier ProvFldDecltorList.
```

```
ProvFldDecltorList= ProvFldDecltor {"," ProvFldDecltor }.

TypeSpecifier     = "void" | "float" | "double" | {"signed" | "
    unsigned" | "char" | "short" | "int" | "long"} | "string" | TypeID
    .
TypeID            = ident.
TypeName          = TypeSpecifier [AbstractDecltor].

ProvFldDecltor    = Declarator.
Declarator        = [Pointer] DirectDecltor.
DirectDecltor     = (ident  | "(" Declarator ")")
                          {DirectDecltorExt}.
DirectDecltorExt  = "(" [IdentList | FormalParamList] ")"
                  | "[" [SliceExpr] "]".

AbstractDecltor = Pointer | [Pointer] DirectAbstractDecltor.
DirectAbstractDecltor = "(" AbstractDecltor ")" | "[" [
    ConstantExpression] "]" | "("[ParamTypeList]")" { "[" [
    ConstantExpression] "]" | "("[ParamTypeList]")" }.
Pointer           = "*" {"*"}.

VariableDecl      = TypeSpecifier Declarator {"," Declarator}.
FormalParamList   = FormalParam {","FormalParam}.
FormalParam       = TypeSpecififer [Declarator | AbstractDeclarator].
IdentList         = ident {","ident}.

ChangeDefinition  = "change" ChangeIdent "/" [ConstantExpr] "/"
    BlockStat.
ChangeIdent       = ident {"::" ident}.
Literal           = ("true" | "false") | string | char | integer |
    float.
Expr              = AssignExpr.
SliceExpr         = ConstantExpr [ ".." ConstantExpr].
ConstantExpr      = LogicalOrExpr.
AssignExpr        = UnaryExpr {"=" LogicalOrExpr}.
LogicalOrExpr     = LogicalAndExpr {"||" LogicalAndExpr}.
LogicalAndExpr    = EqualityExpr {"&&" EqualityExpr}.
EquailtyExpr      = RelationalExpr {"==" RelationalExpr}.
RelationalExpr    = AddExpr {(">"|"<"|">="|"<=")AddExpr}.
AddExpr           = MultExpr {("+"|"-")MultExpr}.
MultExpr          = CastExpr {("*"|"/"|"%")CastExpr}.
CastExpr          = ["("TypeName")"] UnaryExpr.
UnaryExpr         = {"&"|"*"|"!"|"~"|"+"|"-"} PostfixExpr.
PostfixExpr       = PrimitiveExpr ("->" | ".") ident.
PrimitiveExpr     = ident | Literal | "(" Expr ")".

Statement         = VariableDecl ";"
                  | Expr ";"
                  | "if" "(" Expr ")" Statement "else" Statement
                  | "while" "(" Expr ")" Statement
                  | "break" ";"
                  | "continue" ";"
                  | BlockStat
                  .
BlockStat         = "{" {Statement} "}".
```

Listing 5.4: Change Language Syntax

5.4.2.1 Lexical Structure

The grammar in Listing 5.4 describes the individual grammatical rules. These rules consist of terminal and non terminal symbols. The lexical structure further describes the terminals of the language. Terminals are indivisible atoms built from characters.

Key Words.

```
basicblock, bool, break, change, char, continue, double, else, float,
    function, if, int, instruction, long, points, provider, return,
    signed, string, unsigned, void, while
```

Terminals.

```
ident   = (letter | '_') {letter | digit | '_'}.
integer = ('0' ('x' hex {hex} | octet {octet} | {digit}) |
    digitnonzero {digit}).
float   = ('.' | digit {digit} '.') {digit} ['E' [ "+"|"-"] digit {
    digit}].
string  = '"' {stringCh | '\\' printable} '"'.
char    = '\'' (charCh | '\\' printable {hex}} '\''.
```

Identifiers have to start with a letter or an underscore optionally followed by any combination of letter, digit, and underscore.

Binary Operators.

+,-,*,/,%,==,!=,>,>=,<,<=,&&,||,=,.,->,[]

Unary Operators.

&,*,-,+,!,~

Comments.

Starting from '/*' up to '*/', nesting possible and ignoring line feeds.

5.4.2.2 Change Specific Syntactic Aspects

As can be seen in Listing 5.4, the change language resembles a C-like language. It has the concepts of types, variable declarations, statements, and expressions. Additionally a change program has features unique to its usage domain. A valid change program is composed of one or more provider declaration (`ProviderDecl`) and change definitions (`ChangeDef`). A change provider declaration is syntactically similar to a structure declaration in C or a class declaration in C++, C#, or Java.

The following gives a rough overview of the syntactic requirements of a provider declaration:

```
<Provider-Granularity> provider <Provider-Name> {

  /* Valid change points for this provider */
  points { <Change-Point_1>, <Change-Point_2>, ... };

  <ProviderField_1-Type> <ProviderField_1-Name>; // Provider Field
    Declaration
  ...
  <ProviderField_n-Type> <ProviderField_n-Name>; // Provider Field
    Declaration
};
```

The *Provider-Granularity* is one of `module`, `function`, `basicblock`, or `instruction`. The *Provider-Name* has to be a valid identifier. The provider declaration block has to start with a left brace (`{`) and gets terminated by a right brace(`}`). At the end of a provider declaration one can optionally write a semicolon.

Inside a provider declaration block, one can define the valid *provider change points*, followed by *provider field declarations*. The change point declaration is syntactically equivalent to a C language `enum`. The identifier `points` has to be followed by a set of valid change points. Change points have to be inside braces. The list of change point names have to be valid coma separated identifiers.

Provider field declarations follow the C language synatx of:

```
type-name   identifier   ;
```

Provider fields must not have additional parts, like modifiers or initializers. From a syntactic point, change definitions are modeled after function definitions. As for provider declaration, below there is a relaxed syntactic notation of change definitions compared to the EBNF productions in Listing 5.4.

```
change <Change-Identifier>        /* Change Header */
/ <Change-Predicate> /            /* Change Predicate */
{                                 /* Change Body */
  <Change-Statement_1> ;
  ...
  <Change-Statement_n> ;
};
```

Change definitions are introduced by the keyword `change` followed by a valid *Change-Identifier*. Change identifiers are a set of identifiers, separated by double colons (`::`).

After the *Change-Identifier*, the *Change-Predicate* has to be defined. It is delimited from the rest by forward slashes (`/`). The predicate has to be a valid constant expression. The *Change-Body* again follows the C convention and uses braces. Inside the *Change-Body* there are one or more *Change Statements*. As can be seen in Listing 5.4 such a statement can be either a generic type declaration statement, an expression statement, or special statements like `if`, `while`, ...

5.4.3 Semantics

In the last subsection the syntax of the change language was introduced using the grammar in Listing 5.4. Further the grammar elements specific to the domain of writing changes was discussed in more detail. This and the next subsection aim to describe the semantics and context conditions of the change language. Some important semantic aspects of the change language are:

- Value types
- The `string` type
- Pointer types
- Reference types
- Implicit variables
- Predefined types
- Predefined values
- Literals
- Type equality
- Type compatibility
- Assignment compatibility
- Scopes

5.4.3.1 Value Types

Value types have a *pass by value semantics*. If they are passed as parameters, or assigned to other values, the value gets copied.

All primitive types are value types. Primitive types are:

Type	Size [bytes]	Description
`bool`	1	Boolean value. Values: true, false.
`char`	1	Unsigned 8 bit integer value. Represents character literals.
`signed char`	1	Signed 8 bit integer value.
`short`	2	Signed 16 bit integer value.
`unsigned short`	2	Unsigned 16 bit integer value.
`int`	4	Signed 32 bit integer value.
`unsigned int`	4	Unsigned 32 bit integer value.
`long`	8	Signed 64 bit integer value.
`unsigned long`	8	Unsigned 64 bit integer value.
`float`	4	32 bit IEEE floating point value.
`double`	8	64 bit IEEE floating point value.

5.4.3.2 The `string` Type

In C/C++ strings are mutable character arrays, which are terminated by a zero character. This can be inconvenient, since explicit string manipulation functions have to be used to copy or compare strings:

```
const char *a_string = "A String" ;
char *another_string = strcpy(a_string);
if (strcmp(a_string, "A string")==0){     //element comparison
   printf("%s equals 'A string'\n");
}
```

The change language introduces a first class `string` type, that is implemented in terms of the C zero terminated character array concept. If a character pointer is used, the default C reference behaviour is assumed. Otherwise, if string is used, the value type semantic is used:

```
string a_string = "A String" ;
string another_string = a_string;
if ( a_string == "A string") {     //element comparison
   printf("%s equals 'A string'\n");
}
```

This is especially practical when using string comparison in change predicates. Furthermore the change language allows for convenient substring creation. In addition to single element retrieval the array access operator `[]` allows a slice expression. The slice expression is in the form `a..b`. If `s` is a string, the expression `s[a..b]` returns a slice of the string inclusively starting at position `a` until exclusively position `b`.

```
string a_string = "Hello World!";
string hello = a_string[0..5];
string world = a_string[6..11];
```

Listing 5.5: String Slicing Example

5.4.3.3 Pointer Types

The change language has direct support for so called pointer types. Pointer types are composite types. A pointer type always needs another type. For instance a pointer to int (`int*`) is used to store the address of an integer memory cell. If a variable has a pointer type, the variables value is not the value, but is the address to the value. The change language keeps the C language semantics for pointer types. The address-of operator (`&`) and the dereferencing operator (`*`) support pointer operations.

Below you see a code snippet for declaring and using pointer types:

```
int a = 7;
int *pa = &a;          //assign pa the address of a
*pa = 8;               //this changes the value in cell 'a'
printf( "%d\n", a );   //yields: 8
```

Readers interested in learning more about how the C programming language, and hence the change language, deals with pointer types and pointers are referred to [KR78].

5.4.3.4 Reference Types

So called reference types are types that behave like pointer types, but the address is fixed. Reference types as opposed to value types have *pass by reference semantic*. Pointer types have pass by reference semantic as well. For instance on assignment, values are not copied, but are so called aliases to the original value in memory. Implicit provider variables, which are discussed in greater detail below, are reference types.

5.4.3.5 Implicit Values

Some values in the change language are provided by the provider system. These values get implicitly defined.

```
0 change functionProvider::OnEnter
  / functionProvider->functionName == "hello_world" /
2 {
    /* ... */
4 }
```

In the above listing the change definition's predicate on line one uses the variable `functionProvider`. This variable is automatically introduced with the provider declaration. Every change provider creates *an implicit value* that has the same name as the provider identifier. This name can be used in change scripts without declaration. This value is supplied with data from the application's process where the change gets applied. The compiler keeps track of which implicit variables are used and checks whether the usage is valid.

5.4.3.6 Predefined Values

Predefined values are known to the compiler and add functionality to the change scripts. The design is extensible, adding additional predefined values to the application is supported. Currently the following values are predefined:

- `io` value - Writing data back to the monitor process.
- `thread` value - Storing data on a thread local storage.

The `io` value is used to communicate data to the monitor over the communication channel. The change code alters the application's process. Simply printing out values to the standard output would result in displaying the results in the application's standard output instead of writing the output back to the monitor's console.

The `io` value has the member function:

```
int printf(const char *fmt, ...);
```

The following change definition uses the `io` value:

```
change functionProvider::OnEnter
/  /
{
  io->printf( "Entering function %s\n", functionProvider->functionName
     );
}
```

The `thread` value is used to store data on thread local storage. Modern operating systems allow multiple concurrent execution paths inside one process. These are called threads of execution. A thread has its own call stack, but all threads share the memory heap. Thread local storage (TLS) is storage that is relative to each thread. No other thread can directly access the thread local storage of a thread. Per thread call information can be directly associated with one thread. This makes TLS well suited for keeping information related to function calls. For every primitive type the `thread` value has a getter and a setter function:

```
/*
 * get or set a xxxx value on thread local storage.
 */
void setXxxx(const char *name, xxxx value);
xxxx getXxxx(const char *name);
```

For the `int` type the setter would be called `setInt` and the getter would be called `getInt`. The following change definition shows a simple usage of the `thread` and `io` values:

```
0 change functionProvider::OnEnter
  / funcitonProvider->functionName == "say" /
2 {
    thread->setInt("say", thread->getInt("say") + 1);
4 }
```

```
6 change functionProvider::OnLeave
  / functionProvider->functionName == "main" /
8 {
    io->printf("function 'say' was called %d times.\n", thread->getInt("
      say"));
10 }
```

Listing 5.6: Listing: Change definitions using `thread`

For the example application we introduced in Listing 5.1 this would produce the following output at the end of the main function:

```
[monitor-output] function 'say' was called 3 times.
```

The first change definition in Listing 5.6 is called when entering a function called `say`. In line 4 the thread local value named `say` is incremented by one at each. By convention, the `thread` member function `getInt` returns 0, if no variable was already associated with that name. The second change definition beginning on line 7 is called at the end of the program, when the main function terminates. At that point the thread local value `say` is printed using the `printf` member function on the io reference variable.

5.4.3.7 Literals

Literals are tokens in the source code that directly lead to constants in the language. The change language follows the C convention for literals, except that string literals have the value type `string`.

The following literals are supported:

Literal Example	Type	Description
`'c', 'h', 'a', 'R', ...`	`char`	Character literal
`"string"`	`string`	String literal
`1, 2, 0x10, 08, ...`	`int`	Integer literal
`.32, 1.321, 2e07, ...`	`float`	Floating point literal

5.4.3.8 Type Equality

Any value in the change language has a type. Types define the characteristics of values, for instance their size in memory or their structure. Two types are equal, if they are represented by the same `type name`. Pointers are equal if they have the same nested types. The definition for type equality is taken from [Moe03].

5.4.3.9 Type Compatibility

Two types are compatible if one of the following conditions is met:

- Both types are the same.

- Both are pointer types, with the same type qualifiers, that point to compatible types.
- Both are array types whose elements have compatible types. If both specify repetition counts, the repetition counts are equal.
- Both are function types whose return types are compatible. If both specify types for their parameters, both declare the same number of types for their parameters, and the types of corresponding parameters are compatible.
- Both are provider types that are declared in different translation units with the same member names in the same order. Provider type members whose name match are declared with compatible types.

Table 5.4.3.9 shows examples of compatible types:

`type1`	`type2`	Rule
`int`	`int`	Both types are the same.
`long`	`signed long`	Both types are the same .
`char a[]`	`char a[10]`	Both are array types and only one defines a repetition count.
...		

5.4.3.10 Assignment Compatibility

A value of type `Src` is assignable to a value of type `Dest` ($Dest = Src$;) if `Src` and `Dest` are of the same type or if they are implicit convertible to each other. Conversion of a type to another type is often called promotion. The next subsection deals with implicit conversion of values. A special case is the null pointer assignment. `Dest` is a pointer type, and `Src` is an integer with a value of zero. In that sense `Src` is called a null pointer constant. After the assignment the value of `Dest` is called a null pointer.

The listing below shows a null pointer. The value of pointer type `ptrToInt` is referred to as a *null pointer* after the assignment of the null pointer constant (0).

```
int *ptrToInt = 0;
```

5.4.3.11 Implicit Type Conversions

Types define the memory representation of values. Sometimes it is convenient to be able to directly assign a value to another value, even if their types do not share type compatibility. If two values do not have the same type, their memory might be different, and one value can not be simply assigned to another value.

In this case the value of one type has to be converted to the other type. This conversion may not be loss-less all the time. For instance, the following conversion, a valid C code fragment, loses information during conversion:

```
float aReal = 20.99;
int aNumber = aReal;
```

Listing 5.7: Type conversion in the C language

After the assignment in line 2, the value of the variable aNumber will yield 20, while the assigned value is 20.99. Such a conversion of float to int is called a *narrowing conversion*. In most typed languages conversion is performed implicitly by the compiler. When a conversion would lose information, an explicit conversion has to be performed. Less strict languages, like C perform even narrowing conversions implicitly. And some languages, for instance C++, issue warnings, but perform the conversion.

The change language is stricter than the C language with regard to implicit conversions. Implicit type conversions are only allowed if they can not be performed lossless. Otherwise explicit type casts have to be supplied:

```
float aReal = 20.99;
int aNumber = (int)aReal;
```

Listing 5.8: Type Conversion in the Change Language

5.4.3.12 Lexical Scopes

A scope denotes a lexical block of a translation unit, provider declaration, and change definition. Scopes define the visibility of symbols. A variable name is relative to its scope or its parent scope. Thus scopes can be nested. For instance every change definition inherits information from the module scope where it is defined. Predefined values are defined in an common outer scope. This scope can not be directly manipulated within the program language. This outer scope is referred to as *the universe*, as in [Moe03].

5.4.4 Context Conditions

While the syntax only defines what is required for the language to be syntactically correct, many semantic information about the change language were already defined in the last subsection. This subsection covers so called context conditions. These conditions are typically encoded directly in the compiler. Context is additional information that the compiler needs to translate a given source code fragment into a valid target representation.

As an example consider the following change language expression statement:

```
y + 3;
```

If the compiler encounters this expression, it needs to answer, among other things, the following questions:

- What is the symbol referred to by the identifier `y`?
- Is it a variable/constant/...?
- Has the variable already been declared?

- What is the symbol referred to by the literal `3`?
- Is the addition a valid operation on the type of y and 3?
- Are the types compatible?
- What is the type of the expression `y + 3`?
- ...

There are general context conditions, which apply to one or several grammar rules. In this section we will discuss some of the general context conditions as well some specific context conditions concerning change language specific features.

5.4.4.1 General Context Conditions

- In general every name has to be declared before its use.
- Names of predefined values, as defined in Subsection 5.4.3.6, do not need to be declared.
- Every name has to be unique within the same scope.

5.4.4.2 Change Specific Context Conditions

The description of the context specific conditions will focus on the change specific aspects of the language. Every context condition will be discussed per grammar rule. A grammar rule is followed by a description that gives the details of the respective grammar rule's context conditions.

Provider Declaration.

```
ProviderDecl = ("module" | "function" | "basicblock" | "instruction")
   "provider" ident "{" [ProvChangePointDecl ";"] {ProvFieldDecl ";"}
    "}".
```

The provider declaration must match the provider definition running inside the application's process. Provider declarations should be generated by the provider author. Due to performance provider fields are identified by offsets. Any offset mismatch between the provider declaration in a change script and the provider running inside the application's process can result in severe run-time errors.

Change Definition.

```
ChangeDefinition = "change" ChangeIdent "/" [ConstantExpr] "/"
   BlockStat.
```

ChangeIdent must be of the form `<provider-name> :: <change-point-position>`. The provider referred to as `<provider-name>` in the ChangeIdent is the *transforming*

provider of the change. The transforming provider is responsible for evaluating the change predicate and transforming the application's IR if necessary. While a change definition can refer to only one transforming provider, it can access several other providers as so called information providers.

```
change branchInstProvider::Before
/ functionProvider->functionName == "main"/
{
  ...
}
```

Listing 5.9: Change Definition Using a `branchInstProvider`

To compile the above change script the change's transforming provider, called `branchInstProvider`, has to be declared in the same translation unit. The following listing shows a sample provider declaration:

```
instruction provider branchInstProvider {
  points { Before, After };
  int id;
}
```

The predicate expression and the statements of a change definition can access providers in addition to the change's transforming provider. The change definition in Listing 5.9 above uses one additional provider whose declaration is also required in order to evaluate the change predicate.

These additional providers are called *information providers*. These providers need to be declared in the change language script. For instance a partial declaration for `functionProvider` could look like:

```
function provider functionProvider {
  ...
  int functionName;
  ...
}
```

When and how a change is able to access other provider to provide information is explained in more detail below. *ConstantExpr* must be a valid *change predicate*. ConstantExpr has to be of type `bool`. If it is omitted, the value the ConstantExpr defaults to the constant `true`. The provider referenced in the ChangeIdent and all declared providers of a coarser granularity can be used inside the ConstantExpr to define the change predicate.

The following listing shows a set of provider declarations:

```
module provider moduleProvider {
  points { OnEnter };
  int    id;
  string moduleName;
}

function provider functionProvider {
  points { OnEnter, OnLeave };
  int id;
```

```
  string functionName;
}

basicblock provider basicBlockProvider {
  points { AtBegin, AtEnd };
  int id;
  int blockNumber;
}

instruction provider InstrProvider {
  points { Before, After };
  int id;
  string type;
  string category;
  string description;
  bool   isTerm;
}
```

Listing 5.10: Sample Provider Declarations

Given the providers from Listing 5.10 the subsequent fragment would be a legal change definition:

```
// Insert a change code before every
// instruction
change instrProvider::Before
/ moduleProvider->moduleName == "main"
   && functionProvider->functionName == "exampleFunction"
   && instrProvider->category == "memory"
   && instrProvider->type == "malloc"  /
{ ... }
```

The change predicate in the change definition below is not allowed and will result in a compile time error:

```
change basicBlockProvider::AtBegin
/ functionProvider->name == "exampleFunction"
   && instrProvider->type == "malloc"  /
{ ... }
```

The above change is transformed using a basicblock provider. The change's transforming provider is called `basicBlockProvider`. According to Listing 5.10 `basicBlockProvider` is of granularity `basicblock`. Inside a `basicblock` change definition, it is not allowed to use information from providers of a finer granularity. In the above change definition the predicate illegally uses the provider `instrProvider` that is of granularity instruction. The reason for this is that the change framework cannot guarantee that the information of the provider `instrProvider` is valid and useful in this context.

To summarize given the provider declarations of Listing 5.10, the following can be said about access to providers from within the change definitions:

Transforming Provider	*Accessible Information Providers*
`change moduleProvider::...`	moduleProvider
`change functionProvider::...`	moduleProvider, functionProvider
`change basicBlockProvider::...`	moduleProvider, functionProvider, basicBlockProvider
`change instrProvider::...`	moduleProvider, functionProvider, basicBlockProvider, instrProvider

Change Statements.

```
BlockStat = "{" {Statement} "}".
```

Change statements are C-like statements, where normally every name has to be introduced using a variable declaration. There is an exception for references to providers (implicit values) as defined in Subsection 5.4.3.5 and the use of predefined values, as defined in Subsection 5.4.3.6.

For accessing provider references the same restrictions apply to change statements as to the predicate constant expression. The providers accessible to the containing change definition can be used. If we consider a change script with the providers as declared in Listing 5.10, the following change definition would be legal:

```
change basicBlockProvider::AtBegin
/ functionProvider->functionName == "main" /
{
  string moduleName = moduleProvider->moduleName;
  io->printf("first␣block␣in␣main,␣module␣=␣%s\n", moduleName);
}
```

While the following would not be legal:

```
change basicBlockProvider::AtBegin
/ functionProvider->functionName == "main" /
{
  string instructionType = instrProvider->type;
  io->printf("first␣block␣in␣main,␣instructionType=%s\n",
    instructionType);
}
```

The reason why the listing above is illegal is that the change's transforming provider is of granularity basicblock (see the change definition starting with `change basicBlockProvider::...`). This will raise a compile error. Additionally change statements can access any predefined value, as defined in Subsection 5.4.3.6. Currently these are the `io` and the `thread` values. It would make sense to declare predefined values, so that the author could look at the definitions. Future work and enhancements are discussed in Section 8.2.

Some tracing toolkits, like [CSL04], restrict the control flow constructs in the instrumentation language, so that it is not possible to create infinite loops inside the inserted logic. While this is useful, it is more critical in per operating system instrumentation

systems. In [CSL04] the whole operating system could be rendered useless, if an infinite loop would be inserted in the wrong position. The change framework operates within the instrumented application process. Bad or even malicious instrumentation cannot directly influence the whole operating system.

5.5 Change Detection and Recompilation

5.5.1 Overview

In Subsection 5.2.2 the change application process was outlined. In the change framework the application's IR gets transformed by a so called *transforming provider*. Transformation adds nodes to or removes nodes from the IR of the application under analysis. This happens while the application is executing. If the application bytecodes is interpreted this would be all that is needed to incorporate the change into the application. For efficiency reasons the change framework uses a JIT compiling execution engine. This execution engine dynamically compiles the IR to machine code.

In the dynamically compiled mode the application completely runs in machine code. Compilation is done lazily *function at a time*. This means that first the main entry function of the application gets compiled to machine code. During compilation `call` and `invoke` instructions are treated specially. The compiler determines whether the called function has already been compiled or not. If the target function is already compiled, the address of the function is emitted. If not a stub that lazily triggers compilation of that function is added. This way the call graph of the application triggers the translation process.

Calling the stub of a function triggers the JIT compiler to translate the respective function to native code. After that the call site gets redirected to the compiled function. This ensures that next time the machine function is called directly. Application and/or unapplication of changes transform the application's IR. By that the IR gets out of sync with the application's machine code. A mechanism is needed so that the executing machine code detects when it is in need for update.

5.5.2 Recompilation Checkpoints

A recompilation checkpoint in the change framework is a point in the execution of the application, where the code is checked for recompilation. When an application is loaded, the change framework initially transforms the application's IR to contain checkpoints. They are treated as immutable parts, and can not be altered by change application/unapplication. Recompilation checkpoints are used to determine whether the application IR is out of sync with the machine code representation. This is the case after a change was applied by the change framework.

5.5.2.1 Recompilation Detection Period

The recompilation detection period is the time Δt starting from the time a change was applied until the time the application code reflects the change. In the JIT compiler case this is the time that went by between the application IR was transformed and the machine code got recompiled and relinked. Another period, Δp, is the time period between a change is applied, and the change is perceived by the user. Δt has direct influence on Δp.

Δt depends on the following factors:

- Unit of compilation: At which IR node level can be recompiled. The LLVM JIT compiler currently is able to compile individual functions at a time. Recompiling on a finer granularity is not directly supported. This restricts on the way checkpoints have to be used.
- The amount and locations of checkpoints for recompilation: In the LLVM JIT it is reasonable to insert check points before functions are called. Checking on a finer granularity makes no sense since the executing machine function can not be swapped to a new machine function.

Listing 5.11 shows the recompilation checkpoint functionality in a pseudo change definition.

```
change instrProvider::Before
/ instrProvider->type == "call" || instrProvider->type == "invoke"/
{
  if (isApplicationDirty()) {
    ExecutionEngine->recompileAndLinkDirtyFunctions();
  }
}
```

Listing 5.11: Pseudo Change Of Recompilation Checkpoint

A sample application was introduced in Section 5.3.1. Listing 5.1 shows the example application's source code in the C language and Figure 5.4 shows the applicaiton's graphical IR.

In Figure 5.5 one can see the sample application's IR with checkpoints applied. Before any *call* instruction (`call`/`invoke`), a checkpoint tests the change framework's state for a changed IR. If the IR was changed the new IR will be compiled to machine code before the execution of the target function. This ensures that the function call will jump to the new function, if that function was changed.

5.5.3 Checkpoint Testing Overhead

In the current version of the change framework, the checkpoint testing logic consists of a global boolean variable that gets compared. A conditional branch jumps to the recompilation logic if the test is successful. There are two cases. The first case assumes the recompilation flag is false. Hence no transformation happens. In the second case the test for recompilation yields true. Hence recompilation gets triggered. In both cases the following has to performed:

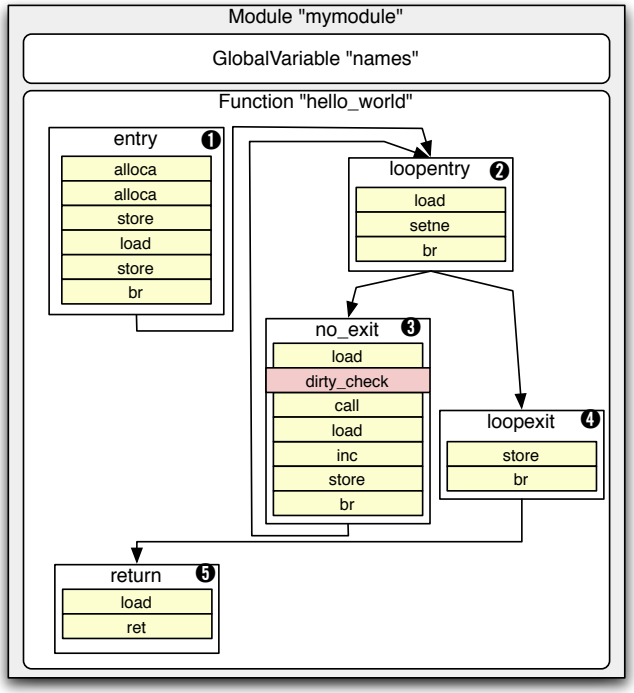


Figure 5.5: Sample Application LLVM IR for Hello World with Checkpoints

- Load the global test variable GV_T into a register R_T - *(Load)*
- Compare the register R_T *(Compare)*
- Conditional jump to a position *(Conditional Changing the PC)*

The steps above have to be executed for every function called in the application under analysis. This means the overall overhead of the change framework depends on how effective this code gets executed.

In the recompilation case additional the subsequent code has to be performed:

- Reset the global test variable GV_T.
- Call the recompilation handler.
- Recompile and relink every dirty function.

In the case that recompilation is needed the time overhead is more absorbable, since the time is dominated by the recompilation time.

Synchronization Issues. Loading and storing the global variable additionally involves synchronization issues to think about. Multiple threads of execution might concurrently write the global flag GV_T. Testing and resetting GV_T has to be done using wait-free synchronization [Her91]. The change registration always sets the value to true, hence this is a single unconditional store instruction which is atomic. The checkpoint code tests the value and sets it to false, if it was set. This is done using a Compare And Swap (CAS) instruction. This ensures that both testing and setting is done by a single instruction. The invariant is that before recompilation is executed the flag is set

to false. This ensures that if a change is applied during recompilation the flag is set to true again.

Coalescing Recompilations. In order to prevent recompiling the same functions too often within a short period of time, the change framework waits a time period t_p before transforming the application's IR. If in that time period, another change arrives, targeting the same function, the two changes are coalesced. Care must be taken, if t_p is chosen too big, the responsibility of the change framework would suffer, hence Δ_P would be too big.

5.5.4 Alternatives to IR Transformation Based Checkpoints

The IR transformation checkpoint approach transforms the initial IR at application load time. The advantage of this approach is that one can easily transform an application to contain checkpoints. The solution is solely based on LLVM instructions, thus no lower level mechanisms needs to performed.

It offers only limited performance and has some other drawbacks:

- It is hard to distinguish between the user authored application logic and the checkpoint logic.
- Checkpoint logic should be transparent, thus it must not be accessible from changes.
- When the application's IR is transformed in a way that obsoletes checkpoints, the checkpoint should be removed.
- When IR is transformed in a way that would require new checkpoints, a checkpoint should be added.

This subsection provides alternative ways for implementing the check pointing logic. Currently check pointing is performed by transforming the applications IR at load-time. One alternative is to implement a `MachineFunctionPass` that is invoked when lowering the LLVM bytecode to machine code. Another way would be to extend the LLVM bytecode or to add a so called intrinsic function.

Both options add the benefit of making check points transparent to the rest of the bytecode transformation process. The downside of both alternatives is that they are more low level and therefore have to be explicitly ported to other targets.

5.5.4.1 Write a LLVM MachineFunctionPass

The LLVM compiler infrastructure makes heavy use of *Passes*. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code[LLV06d]. A pass manager keeps track and schedules a set of passes that work on a specific granularity and transform the LLVM IR. Code generation is done by creating so called `Machinefunction`s out of LLVM function nodes. Later an `Emitter` writes the machine code into a block of memory or a file.

A `MachineFunctionPass` operates on individual `MachineFunction`s. Every LLVM IR function node has one corresponding `MachineFunction`. When the JIT is created it initializes the code generator. Among other things this is the point where the passes are added to a so called pass manager (`FunctionPassManager`).

In pseudo code a `MachineFunctionPass` to emit checkpoints could look like:

```
/* This is target machine specific psuedo code, that
   outlines the tasks needed to add a checkpoint
   before a function is called. */
procedure runOnMachineFunctionX86(machineFunction:MachineFunction)
begin
  for each machineBlock ∈ machineFunction.basicBlocks
  begin
      for each machineInstr ∈ machineBlock.instructions
      begin
         /* if we are in a callsite, then emit the
            check pointing code in a machine specific
            way. */
         if machineInstr.opcode ∈ {CALL, TAILCALL} then
            Add code for checkpoint before machineInstr.
         end
      end
  end
end
```

After the JIT registers the checkpointing function pass, the checkpointing pass is performed for each call to compile a function to machine code. The advantages are:

- The checkpoints are not visible inside the applicaiton's IR.
- Checkpoints are compiled on an as needed basis.

`MachineFunctionPasses` do not translate the application's IR, but build a separate machine level model. Hence the recompilation checkpoints are invisible at the bytecode level. This makes the change application easier, since there are no instruction sequences that have to be treated special.

The `MachineFunctionPass` provides access to all the target machine specific functionality, but has the drawback that a `MachineFunctionPass` has to be implemented per target machine to make sense.

5.5.4.2 Implement an Intrisic Function or a Custom Bytecode

Being an infrastructure for compiler engineering, the LLVM instruction set is not untouchable as for instance the bytecode of high level virtual machines, like the CIL, or the Java bytecodes. Extensions to the LLVM instruction set can be done. Adding instructions is seldom needed and introduces binary compatibility issues. Often a better way is to introduce a so called intrinsic function. Intrinsic functions are treated as ordinary function calls. The interesting feature of intrinsic functions are that they are known to the compiler.

In this subsection we will consider the case of implementing an intrinsic function. Introducing a custom bytecode has no additional benefit, interested readers are referred to [LLV06b] for details on adding a custom bytecode to the LLVM infrastructure. Intrinsic functions are identified by name. An intrinsic function name has to start with a `llvm.` prefix. No other function names are allowed to have that prefix. Please see Section *Intrisic Functions* in [LLV06c] for details.

Lowering an intrinsic means that the high level intrinsic concept is translated to lower level instructions. In the simplest case an intrinsic function is treated as an ordinary function.

Generic Lowering of Instrinsics. Intrinsic functions are normally handled by the code generator. If the code generator cannot handle an intrinsic, or does not know about the intrinsic, the pass `LowerIntrinsicCall` can be used to transform the intrinsic call in a target independent way. For instance the intrinsic `llvm.memcpy` can be implemented as a call to the CRT library's `mempcy` if the target machine has no hardware support for copying multi words, or if it has not been implemented yet.

Target Specific Lowering of Intrinsics. Implementing the target specific lowering involves translating the high level intrinsic function call to machine code. In that case an intrinsic is not much different from a custom bytecode. The generic `SelectionDAGLowering` instruction visitor has a visitor method called `visitIntrinsicCall` as well as a target specific one called `visitTargetInstrinsic` that defers lowering.

The intrinsic function for checkpoints could be named `llvm.checkpoint`. The implementation is twofold:

- A target independent version that is converted to LLVM instructions.
- A target dependent version that makes heavy use of target dependent features.

A big advantage of this option is that an intrinsic function call does not introduce new basic blocks. In the case of LLVM bytecode translation, inserting a conditional jump instruction implies changing the CFG of that function.

5.5.5 Finer Recompilation Models

In this subsection the issue of testing for recompiling on a finer level than per function is revised. Recompiling on a level finer than function is also a big issue especially in the field of adaptive dynamic compilers. The following C application is used as an example in this subsection:

```
void f1() {
   printf( "hello␣from␣f1\n" );
}

void f2() {
   printf( "hello␣from␣f2\n" );
}
```

```
/* main entry point into the application */
void start_app( ) {
  /* infinite loop */
  while ( 1 )
  {
    f1();
    f2();
  }
}

int main() {
  start_app();
}
```

The problem is that the loop inside `start_app` is never exited. Thus the `start_app` function is only called once, but executes throughout the whole life of the application.

Applying the following change (using the provider declaration as of Listing 5.10):

```
change instrProvider::Before
/ instrProvider->type == "call"/
{
   io->printf( "func: %s: instr: %s\n",
         functionProvider->functionName,
         instrProvider->description
         );
}
```

With the per function recompilation model the output should look like the following:

```
[monitor-output] func: f1, isntr: call printf
[monitor-output] func: f2, isntr: call printf
[monitor-output] func: f1, isntr: call printf
[monitor-output] func: f2, isntr: call printf
[monitor-output] func: f1, isntr: call printf
...
```

The `start_app` function never gets recompiled, because it is not called anymore than at the beginning. Additionally adding checkpoints when entering loops and after basicblocks, that are frequently re-executed, can lower the recompilation detection time period Δt. When performing checks on a finer level than before function calls one has to switch to the new machine code while the old machine code is executing. For example after performing a recompilation in a loop header, the function currently executing is still the old one. The new code never gets executed this way.

In this situation a migration of a running function to a new one is needed. The trickiest part in migrating an executing function is to migrate the function's activation record. One technique that performs this is called on-stack-replacement (OSR). OSR was pioneered in the Self programming language implementation [Hoe94]. Dynamically replacing methods in Java is discussed in Chapter 7 of [Dmi01]. Of interest are Chapter 7.3.1 "Dealing with Active Methods" as well as Chapter 7.5.3 "Multiple Policies for Dealing with Active old Methods of Changed Classes". Chapter 7.3.1 of [Dmi01] specifies one method that is interesting for the change framework:

> "... Identify a point in the new method that corresponds to the current execution point in the old method, and switch execution straight from the old method code to the new one. In certain cases, e.g. when a method being evolved never terminates, and the changes are purely additive and free of side effects (for example, trace printing statements are added), this can be the desired and useful semantics. However, in more complex cases it may be very hard for the developer to understand all of the implications of a transition from one code version to another at some arbitrary point. One other application of the mechanism developed for this policy may be for dynamic fine-grain profiling ..."

As can be seen above dynamically replacing running functions or methods is an ambivalent topic. On the one hand does it provide very responsive instrumentation results, on the other hand does it lead to undesired side effects, especially if the change is more complex. A mixture of bytecode instrumentation and so called code patching would serve as a good approach for performing whole function replacement. Please refer to [Dyn06] for a code patching framework.

5.6 Change Protocol

5.6.1 Overview

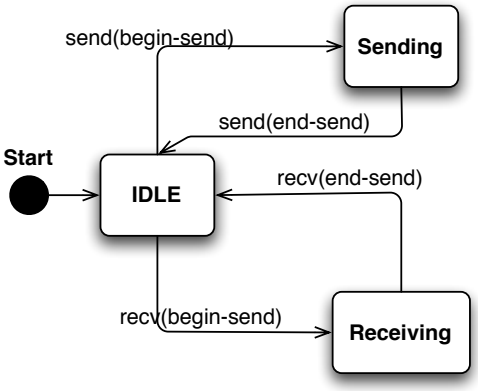


Figure 5.6: Fundamental Channel State

As shown in Figure 2.1 the change framework runs across two distinct processes. One is the application's process the other the monitor process. Between these two processes there is communication channel. Every tuple of $(monitor, application)$ share one channel. While an application process might have many monitors and a monitor process might control many different applications, every two processes only use one channel.

Some general objectives of the communication system are:

- Only one bidirectional channel should be needed between one monitor and one application process.

- The system should work over the Transmission Control Protocol (TCP).
- IO overhead should have almost no effect on the monitored application's performance.

Figure 5.6 shows the channel states. Every communication channel has two end points. Every end point is in one of the states, shown in the figure. If one end is in the sending state the other endpoint is in the receiving state and vice versa.

5.6.2 Communication Framework

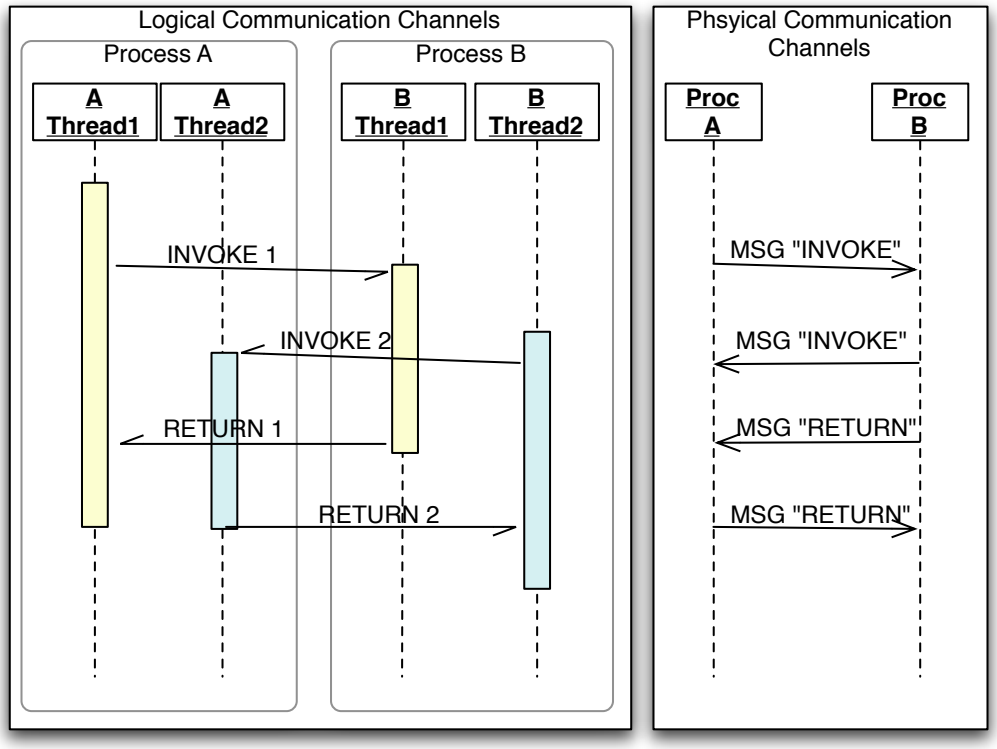


Figure 5.7: ChangeIO Communication Framework

The change protocol channel maps many logical channels to one physical channel. This is done by turning blocking bidirectional sends into non blocking one-way sends. Figure 5.7 shows two types of communication channels. The left rectangle shows the logical communication. The right rectangle shows the physical communication channel. In the logical representation every two threads have their own blocking channel. In the logical channel two processes share one channel. Every physical communication is done via a one-way message.

5.6.2.1 Channel Message Handling

The messages are processed in so called *nonblocking* mode. The channel API provides a blocking abstraction that waits when `INVOKE` messages are sent until a matching `RETURN`

message is received. This is just for convenience as all messages are internally processed in nonblocking mode.

The `Channel` class partially outlined in pseudo code below provides the API for sending and receiving messages over the communication channel. The communication channel can be any bidirectional reliable connection. This means the connection must support read as well as write in both directions. In the current implementation the communication channel is carried out over a TCP or UNIX domain socket connection.

```
class Channel
  ...
```

The `receivedReturns` is a queue, for storing received return messages. It has a specified capacity. If the queue is full the oldest entries are dropped. The `waitList` keeps track of threads waiting for receiving return messages. The `ChannelHandle` abstracts the concrete implementation of the IO operations.

```
var receivedReturns : LookupQueue[integer, ReturnMsg]
var waitList : HashMap[integer, Thread]
var handle   : ChannelHandle
...
```

The `sendOnewayInvoke` method of the channel represents sending of simple oneway messages.

```
method sendOnewayInvoke( msg: InvokeMsg )
begin
   sendMsg(msg)
end
```

On the other hand the `sendInvoke` method of the channel mimics a blocking invocation. The message is sent and the calling thread is put to sleep until a matching return message is received:

```
synchronized method sendInvoke( msg: InvokeMsg )
begin
  sendMsg(msg);
  return waitForReturn(msg.id);
end
```

The `waitForReturn` method blocks the calling thread until a return message with the requested id is received. For this the Channel needs to keep track of listening threads to notify, if a ReturnMsg is received. Additionally, the ReturnMsg are kept in a queue. The queue has a specific max capacity. If the capacity is full, the oldest entry is dropped.

```
synchronized method waitForReturn(id : integer, timeout : long) :
 ReturnMsg
  var returnMsg : ReturnMsg;

begin
  if receivedReturns.find(id) != nil then
     return receivedReturns.remove(id)
  end
  waitList.at(id) := Thread.currentThread()
```

```
    while true
    do
      try
          Thread.currentThread().wait(timeout)
      except InterrupedException => break
      except Timeoutexception => raise Error("Did not receive an
    answer within the timeout period.")
      except Throwable => continue
      end
    end

    waitList.remove(id)
    returnMsg := receivedReturns.at(id)
    raise Error if returnMsg == nil
    return returnMsg
  end
```

When a `RETURN` message is received by the system, the channel's `onReceiveReturn` method is invoked. This method first adds the ReturnMsg to the list of returns. Then it looks whether there is a thread already waiting for this return. If yes, the thread that is blocking is waked up.

```
  synchronized method onReceiveReturn( msg : ReturnMsg )
  begin
    receivedReturns.add(msg)
    if waitList.contains(msg.id) then
        waitList.at(msg.id).interrupt()
    end
  end

end Channel;
```

5.6.3 General Message Wire Format

Listing 5.12 shows the fundamental message type. The data types `u1`, `u2`, and `u4` represent an unsigned one-, two-, or four-byte quantity, respectively. Every message starts with a tag byte. The tag byte indicates the type of the message. The tag is used to interpret the rest of the message.

Message tag	Value	Description
`INVOKE`	1	This message indicates an invocation. When the client has finished processing this message, it will send a `RETURN` message.
`INVOKE_ONEWAY`	2	The message is a oneway message. The receiver does not need to send anything back when finished with processing.
`RETURN`	3	The message indicates the sender of an `INVOKE` message that the callee has finished processing the message. The `RETURN` message's `id` has to match the `id` of the `INVOKE` message.

```
MSG {
  u1   tag;
  u4   id;
  u4   type;
  u1   data[];
}

enum MSG_Tag {
  INVOKE = 1,
  INVOKE_ONEWAY = 2,
  RETURN = 3
}
```

Listing 5.12: Message Payload

5.6.3.1 Invoke Messages

`INVOKE` messages are sent from the caller to the callee to pass data from the one side of the channel to the other. `INVOKE` messages are identified by the `INVOKE` `tag` as shown in Listing 5.12. The `type` field of the message defines the type of `INVOKE` message that the special message represents. The type field is used to interpret the message. Hence the message is not self hosted. The other side needs to know what serialization format the specific type conforms to.

5.6.3.2 Return Messages

`RETURN` messages always have a matching `INVOKE` message. A `RETURN` message is sent by the callee after processing of the invoke message completes. The return message payload contains the type of the `RETURN` message. One is indicating success, the other is indicating failure.

Status Class	Representation
`SUCCESS`	$is\_success(code) := code >= 0$
`ERROR`	$is\_error(code) := code < 0$

The status code is a signed 32 bit integer, using the twos complement representation. A value of zero or greater denotes success. Negative values denote the processing of the `INVOKE` yielded an error.

The following shows the layout of the `RETURN` message. The payload is passed as an opaque array of bytes. The payload might be return values such as primitive types or record values.

```
RETURN_MSG {
  u1   tag;
  u4   id;
  u4   type;
  u4   status;
  u1   data[]:
}
```

5.6.3.3 Oneway Invoke Messages

A special type of message is the `INVOKE_ONEWAY` message. The wire representation of `INVOKE_ONEWAY` is identical to the `INVOKE`. These messages are especial useful for providing a fast means to deliver data from one process to the other, without having to wait for a return message. As we will see later the `IO_OUTPUT` message makes use of this technique to provide fast processing of IO data.

5.6.4 Well Known Change Specific Messages

The last section introduced the general communication framework, which consists of:

- The `Channel` abstraction
- The principal asynchronous message types: `INVOKE`, `INVOKE_ONEWAY`, and `RETURN`
- An abstraction for blocking `INVOKE` calls, that wait for `RETURN`.

This subsection shows the change specific message types, which are based on the communication framework. As can be seen in Listing 5.12 the MSG representation contains a 32 bit type information field. This type information is mapped to message types. The following predefined messages are currently supported:

- `REGISTER_CHANGE`
- `UNREGISTER_CHANGE`
- `IO_OUTPUT`
- `CLOSE`

5.6.4.1 `REGISTER_CHANGE` Message

Description This message is sent from the change monitor to the application's process in order to register a change with that application. It is composed of a `INVOKE` and a `RETURN` message:

- The `INVOKE REGISTER_CHANGE` message, sent from the monitor to the application.
- The `RETURN REGISTER_CHANGE` message, sent from the application to the monitor.

The message contains the compiled change script and change meta data. The return message contains the unique change identifier and a cookie for reference to the registered change. The following shows the wire representation of the `REGISTER_CHANGE` invoke as well as the return message.

```
REGISTER_CHANGE_INVOKE_MSG
{
  u1  tag;                                  // INVOKE
  u4  id;
  u4  type;                                 // REGISTER_CHANGE
  u4  change_info_count;
  u1  change_info[change_info_count];       // Meta data of the change
  u4  change_module_count;
```

```
  u1  change_module[change_module_count]; // LLVM bytecodes of the
    change
}

REGISTER_CHANGE_RETURN_MSG
{
  u1  tag;                                // RETURN
  u4  id;
  u4  type;                               // REGISTER_CHANGE
  u4  status;
  u4  change_cookie;                      // Registration cookie
  u1  change_id[16];                      // Change ID
}
```

5.6.4.2 `UNREGISTER_CHANGE` Message

Description This messages is sent from the change monitor to the application's process in order to unregister a change with that application. It is composed of 2 message sends:

- `INVOKE UNREGISTER_CHANGE` message, sent from the monitor to the application.
- `RETURN UNREGISTER_CHANGE` message, sent from the application to the monitor.

The following shows the wire representation of the `UNREGISTER_CHANGE` invoke as well as the return message.

```
UNREGISTER_CHANGE_INVOKE_MSG
{
  u1  tag;                                // INVOKE
  u4  id;
  u4  type;                               // UNREGISTER_CHANGE
  u4  change_cookie;                      // Cookie referring to the
    change
  u1  change_id[16];                      // Change id (UUID)
}

UNREGISTER_CHANGE_RETURN_MSG
{
  u1  tag;                                // RETURN
  u4  id;
  u4  type;                               // UNREGISTER_CHANGE
  u4  status;                             // Error code
}
```

5.6.4.3 `IO_OUTPUT` Message

Description This messages is sent from the application's process to the change monitor if output is to be redirected to the monitor. It is a oneway message. `IO_OUTPUT` invoke messages do not have to be followed by return messages.

The change script below results in `IO_OUTPUT` for every function entered:

```
change functionProvider::OnEnter
/  /
{
  io->printf("Entering %s\n", functionProvider->FunctionName );
}
```

The following shows the wire representation of the `IO_OUTPUT` oneway invoke message.

```
IO_OUTPUT_INVOKE_ONEWAY_MSG
{
  u1  tag;                                  // INVOKE_ONEWAY
  u4  id;
  u4  type;                                 // IO_OUTPUT
  u4  data_count;
  u1  data[data_count];                     // IO_OUTPUT Payload
}
```

5.6.4.4 `CLOSE` Message

Description The close message closes the current session between the application and the monitor process. It can be either sent by the application or by the monitor. The `CLOSE` message, if sent from the monitor to the application can contain optional information for unapplying all the changes that client has registered with the server. This can be handy if the monitor wants to make sure that all the changes that were registered, are undone.

If a change remains in the application after the session is closed, further calls performed on the `io` value will result in no-ops since the communication channel will be closed. Normally the `CLOSE` invoke message has an accompanying return message, but it can also be sent as a oneway message if the sender does not need the outcome of the call. In order to make sure that the close message was received by the other end of the channel, it is encouraged to process the return message.

```
CLOSE_INVOKE_MSG
{
  u1  tag;                                  // INVOKE
  u4  id;
  u4  type;                                 // CLOSE
  u4  cookie_count;                         // Number of changes to be
    unapplied
  u4  cookies[cookie_count];                // Changes to be unapplied
}

CLOSE_RETURN_MSG
{
  u1  tag;                                  // RETURN
  u4  id;
  u4  type;                                 // CLOSE
  u4  status;
```

```
    u4  unapplication_count;                    // Number of change
      unapplication
}
```

This section introduced the change protocol. The change protocol defines both the logical and physical channel abstraction. In addition the low level messages have been defined. The last part showed well known message types related to the change framework.

5.7 Summary

This chapter introduced the change framework. First an overview was presented. Then the overall architecture, including the change concept, and the application and unapplication of changes was outlined. The framework consists of two processes. The application under analysis as well as a monitor process. The infrastructure of the change framework is the LLVM compiler infrastructure.

After the framework architecture the concept of change providers were introduced. Change providers provide static context information and the ability to conditionally transform the intermediate language at specific change points. This can be used by change scripts. Change scripts are written in the change language and get compiled to LLVM bytecodes by the monitor process.

In the forth section the change language was defined. The change language is a C like language with domain specific support for expressing changes. After introducing the change language, change detection and recompilation was introduced. After a change is applied or unapplied the intermediate representation has to be recompiled and relinked in order for the application to reflect the changes.

In the last section the change protocol was introduced. The change protocol abstracts the I/O between a monitor process and an application process running the change framework. First an overview over the communication framework was provided. After that the general message wire format was given. Eventually well known change specific messages were presented.

To summarize the change framework is a flexible system for dynamically transforming applications. Important properties of the change framework are:

- Dynamic application/unapplication of changes.
- Reversibility of changes.
- Extensibility through change providers.
- Interactivity through iterative application/unapplication of changes.
- Portability through transformation on the intermediate level.
- Network transparency through a communication framework.

Chapter 6

Evaluation

6.1 Overview

In this chapter, the change framework will be evaluated. The first part will be a general analysis of the instrumentation done. The outcome of this evaluation will the theoretic expectations of the change framework.

In the second part the existing prototype of the change framework will be evaluated. First the LLVM framework's execution engine will be discussed and evaluated. Further the protoype implemenation will be discussed, based on the figures and numbers obtained by the LLVM evaluation.

6.2 Analysis of the Change Framework

In the machine code execution model, the program representation can be directly executed by the available processor. On GNU/Linux running on an Intel IA32 architecture, an executable application `helloworld` compiled to *i386* and represented in the executable and link format (ELF) can be directly executed. The GNU compiler collection (GCC) maintains a unique target string that identifies the processor and the operating system.

The change framework is a system for dynamic program instrumentation and anlysis. Such a framework allows one to reason about and change a program at run-time. While there exist frameworks that instrument machine code executables [Dyn06], [CSL04], the basis of the change framework is the higher level LLVM representation.

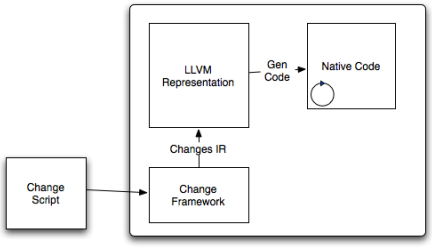


Figure 6.1: Change Framework Overview

As can be seen in Figure 6.1 one key aspect of the change framework is that the LLVM representation remains in memory during run-time and that change application first transform the LLVM representation, which then forces the execution engine to recompile and relink machine code. This is required in order to remain machine independent and to view the program as a set of LLVM artefacts. While there exist instrumentation and code patching frameworks that work on the machine code level (such as [Dyn06]) these require additional information such as debug information. Otherwise these frameworks would not be able to reproduce information about the source level program, such as type information, variable declarations, and much more. The benefit of richer application semantics encoded in the LLVM representation comes at the cost of execution time and space requirements.

6.2.1 Analysis of LLVM

LLVM as opposed to higher level virtual machines does not require additional run-time information on the application data structures itself. The memory requirements of the program itself does not increase. The difference between the machine code compiled application and an application dynamically executed using the LLVM execution engine is that the LLVM bytecodes are not directly executable by the processor. Hence the LLVM application needs both representations in memory, the LLVM bytecodes as well as parts of the machine code representation. The processor executes the machine code.

The memory requirements of a dynamically executed LLVM application amounts in the data memory requirements of a statically compiled application plus the memory requirements of the code and data of the just-in-time compiler. The data requirements of the compiler are especially the code buffers from which the application gets executed from. The reason why the memory overhead of the application's data is the same regardless of the compilation model, is that the data structures are layout equivalent in both the static compiled GCC and the LLVM compiled version. More formally let M be the memory requirement of an application. M of the machine code application is defined by the requirements of the $Code$ and the requirements of the $Data$ of that program:

$$M(\mathscr{P}_{machinecode}) = Code(\mathscr{P}_{machinecode}) \cup Data(\mathscr{P}_{machinecode})$$

Any program $\mathscr{P}$ in machine code form consists of its machine code, and its data. A program in LLVM representation, if dynamically executed, needs another program $\mathscr{P}_{JIT}$, which dynamically translates the LLVM representation to *machinecode* representation. The application code's data requirements are equal between an application in *machinecode* and LLVM representation.

The following states that the LLVM version of a program has exactly the same data structures as the machine code application code:

$$Data(\mathscr{P}_{machinecode}) = Data(\mathscr{P}_{llvm}) = Data(\mathscr{P})$$

Hence the following formula gives the additional overhead of a program in LLVM representation, which is executed by $\mathscr{P}_{JIT}$:

$$M(\mathscr{P}_{JIT}) = Code(\mathscr{P}_{JIT}) \cup Data(\mathscr{P}_{JIT})$$
$$M(\mathscr{P}_{llvm}) = M(\mathscr{P}_{JIT}) \cup Data(\mathscr{P}_{llvm})$$
$$M(\mathscr{P}_{llvm}) = Code(\mathscr{P}_{JIT}) \cup Data(\mathscr{P}_{JIT}) \cup Data(\mathscr{P})$$

An application $\mathscr{P}$ in LLVM representation ($\mathscr{P}_{llvm}$) to be executed by $\mathscr{P}_{JIT}$ has the memory requirements of $\mathscr{P}_{JIT}$ plus the data of $\mathscr{P}$. The performance of an application is defined by many properties. Code quality and solid software engineering on the one hand, as well as a good optimizing compiler can both improve the run time of the application. An application $\mathscr{P}$ in LLVM representation that is binary translated by $\mathscr{P}_{JIT}$ to a *machinecode* representation on a just in time basis, basically consists of two applications, namely $\mathscr{P}_{JIT}$ and $\mathscr{P}$.

Just-in-time (JIT) translation means that a function if executed for the first time gets translated to *machinecode* and then the machine code gets executed. If the function gets called later the already compiled machine code gets executed. If a function is only called once, the overhead imposed by $\mathscr{P}_{JIT}$ is relative high, whereas if a function is called very often, the overhead might be negligible.

The execution time T of a dynamic compiled application in $LLVM$ representation can be decomposed into:

$$T(\mathscr{P}_{llvm}) = T_{Exection} + T_{Translation}$$

$T_{Exection}$ is the time the applicaiton spends executing the application code, while $T_{Translation}$ is the time the virtual machines spends compiling the bytecodes to machine code. $T_{Translation}$ only occurs for every function executed for the first time. The evaluation section observes $T_{Translation}$ for two bytecode representations. One is a version with heavy function inlining enabled. The other does not do inlining at all.

6.3 Evaluation of the Existing Prototype

Part of this thesis was the implementation of a prototype of the change framework. The prototype originally was based on LLVM version 1.5, released in May 2005. The LLVM framework, at the time of this writing, is in version 1.9. Because there where many stability improvements between these two versions, version 1.9 will be used for evaluation purposes.

This section is organized as follows. First the infrastructure is evaluated and the differences between a set of C programs, when executed as either a machine code executable compiled by GCC, as a machine code executable compiled by the LLVM ahead of time compiler, as a heavily inlined bytecode module executed by the JIT base, or as a non inlined bytecode module executed by the JIT, are shown.

6.3.1 Evaluation of the LLVM Infrastructure

Evaluation of the LLVM infrastructure was done by extending the existing LLVM testing infrastructure. The evaluation suite consists of a set of applications written in the C programming language and compassing a single source file.

6.3.1.1 Common LLVM Suffixes

The LLVM infrastructure embeds itself seamlessly into the machine code toolchains. It is often hard to differentiate various intermediate files, like machine code assembly, machine code object files, LLVM assembly, or LLVM bytecode files. Here is a list of suffix as well as their meaning in our evaluation.

- `.s` - Machine Code assembly. If the target is i386, this is IA32 assembler in AT&T assembly syntax.
- `.o` - Machine Code object files. This files are individual translation units in machine code. One or more `.o` files are linked together to one library or executable.
- `.ll` - LLVM assembler. This is the textual representation of LLVM.
- `.bc` - LLVM bytecode module. This is a program in binary LLVM representation. This files can be dynamically executed using the interpreter or JIT. `.bc` files can both represent a single translation unit, as well as a whole linked program.

6.3.1.2 The Test Suites

Each C program belongs to one of the following test suites:

- Coyote Benchmark
- Dhrystone
- McGill
- ShootOut

The Coyote Benchmarks perform numeric calculations, data compression, and more. According to the description of [Coy04] the benchmark is still in development. The following tests were used:

- *alma* - Calculates the daily planetary ephemeris (at noon) for the years 2000-2099; tests: array handling, floating-point math, and mathematical functions such as `sin()` and `cos()`.
- *huff* - Compresses a large block of data using the Huffmann algorithm; tests: string manipulation, bit twiddling, and the use of large memory blocks.
- *lp* - A number-crunching benchmark that can be used as a fitness test for evolving optimal compiler options via genetic algorithm.

The next test suite is the Dhrystone test suite. The following programs are part of the Dhrystone suite:

- *dry* - Contains statements of a high-level programming language (C) in a distribution considered representative: assignments - 53 %, control statements - 32 %, procedure, function calls - 15 %.
- *fldry* - Similar to the *dry* test, but uses floating points instead of integer data types.

The so called McGill test suite consists of the following test apps:

- *chomp* - Solves a simple board game.
- *exptree* - Given a set S of k nonzero numbers, labeled $n_1...n_k$ and a total $t \in N$. Find an expression tree, whose leaves are taken from S, and whose interior nodes are each labeled $\in \{+, -, *, /\}$, and that evaluates to t. There is the additional restriction that any subtree must evaluate to a natural number greater than 0. The program uses a brute force search, more precisely an iterative deepening depth-first search.
- *misr* - This program creates two multi input shift registers (MISR's) one which contains the true outputs and the other in which the outputs are not corrupted with the probability given in the input.
- *queens* - This program finds all the possible ways that N queens can be placed on an $N \times N$ chess board so that the queens cannot capture one another, so that no rank, file or diagonal is occupied by more than one queen. This is an example of the utility of recursion. The algorithm uses recursion to drastically limit the number of board positions that are tested.

The ShootOut test suite is called the *Programming Language Shootout*, it encompasses the following test cases:

- *ackermann* - Calculates the Ackermann series.
- *ary3* - Array test.
- *fib2* - Calcuates the Fibonacci series.
- *hash* - Creates string hashes.
- *heapsort* - Sorts a random array using heap sort.
- *hello* - Simply prints `'hello world'`.
- *lists* - Performs mutations on a double linked list.
- *matrix* - Performs matrix arithmetic.
- *methcall* - C program that emulates object oriented features.
- *nestedloop* - Iterates over 6 levels of for loops.
- *objinst* - C program that emulates object oriented features.
- *random* - Generates many random numbers.
- *sieve* - The Sieve of Eratosthenes for finding prime numbers.
- *strcat* - String concatenation.

6.3.1.3 The Run-Time Environments

Every test application `${APP}` was automatically compiled into the following targets:

- `${APP}.machine` - Machine Code executable compiled with the GCC compiler using `-O2` flags.
- `${APP}.llvm.machine` - Machine Code executable compiled with the `llvm-ld` out of bytecode.

- `${APP}.llvm.bc` - LLVM bytecode module file, compiled using `llvm-gcc` and `gccas` utility.
- `${APP}.noinline.bc` - LLVM bytecode module file, linked with the `-disable-inlining` flag. The LLVM just-in-time compiler performs no further inlining.

The machine code applications are executed directly. The bytecode files are executed using the LLVM just-in-time execution engine (`lli`).

6.3.1.4 Testing Infrastructure

The test machine is an Apple MacBook Pro:

Computer	MacBook Pro 15"
Computer Model	MacBookPro (1.1)
Processor	Intel Core Duo
Processor Speed	1.83 GHz
Number of CPUs	1
Number of Cores	2
L2-Cache (per CPU)	2 MB
Main Memory	1 GB
Bus Speed	667 MHz

The operating system and system software information is:

Operating System	Mac OS X 10.4.8 (8L2127)
Kernel Version	Darwin 8.8.1
GCC Version	GCC version 4.0.1 (Apple Computer, Inc. build 5250)
LLVM Version	LLVM v 1.9

6.3.1.5 Comparing Overall Run Time

The total run time of the programs is measured using the `time(2)` UNIX program. Figure 6.2 shows the overall run time of the individual benchmarks, compared among the four targets:

Figure 6.2 shows that the statically compiled LLVM version is often the fastest. The exptree test compiled with `llc` was about 15 times faster than the version compiled using GCC with medium optimization turned on. Especially for short running applications, the just-in-time overhead shows. The shorter the run time of the program is, the greater is the relative impact of the translation time ($T_{Translation}$).

The *ackermann* test only goes two recursions deep. In this test $T_{Exection}$ is very small. Here one can see that the impact of $T_{Translation}$ is visible. It can be seen the relationship between the `llc` and the `lli` compilers. For instance in the *dry* test, every LLVM compiler outperformed the GCC compiler around 5 times. Here the non-inline version has the worst speedup. In the *sieve* and the *hufbench* application the GCC is faster than the LLVM compilers. Overall it can said that, if the LLVM compiled version is faster, it is significantly faster than the GCC compiled application. The negative impact of $T_{Translation}$ does not show in most but very short tests.

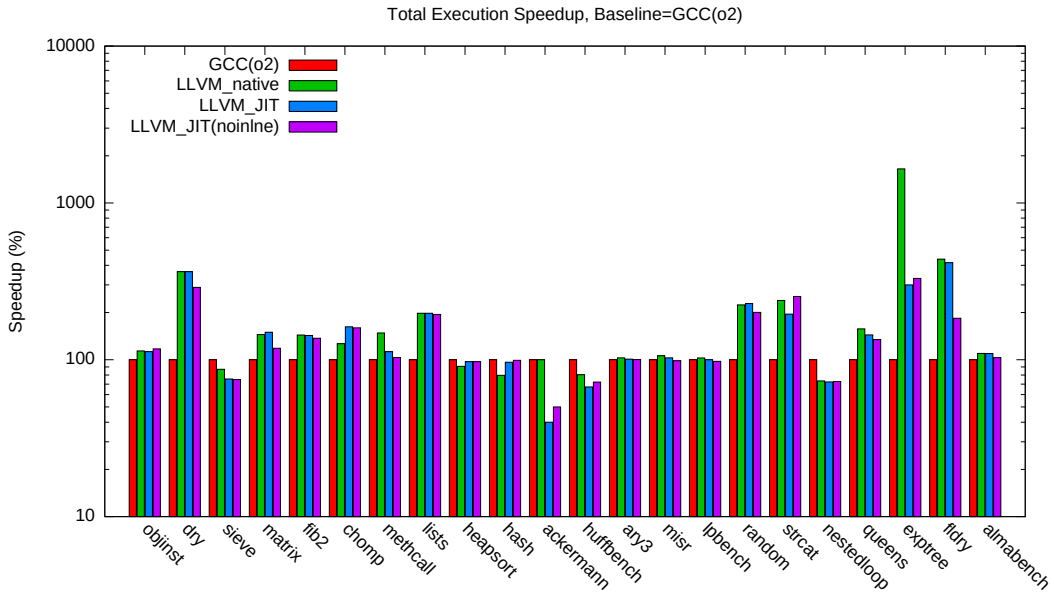


Figure 6.2: Execution Speedup Compared to the GCC(o2) Version (Higher bars are better)

6.3.1.6 Translation Time vs Overall Execution Time

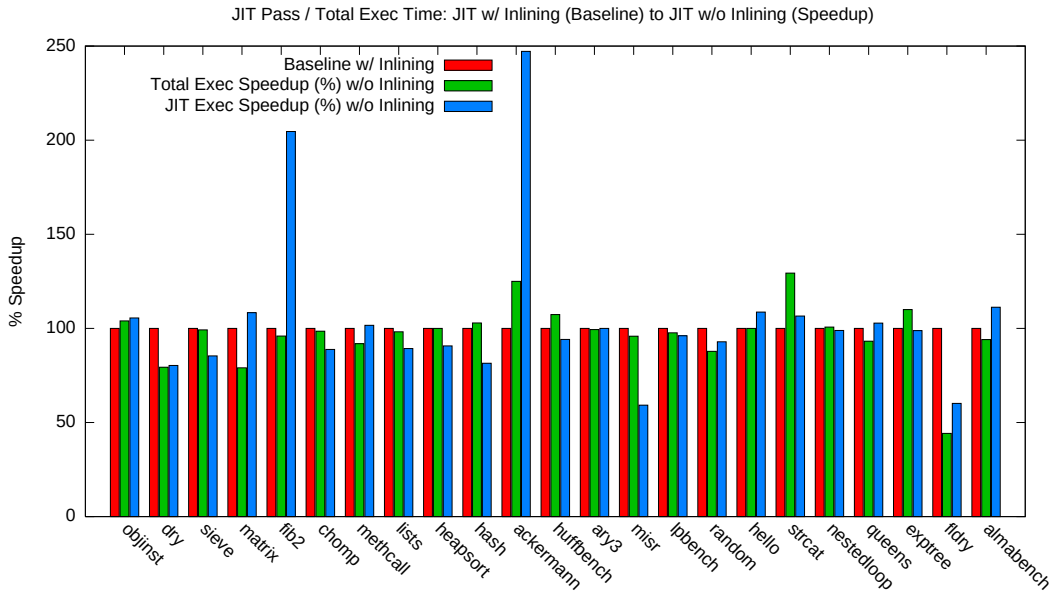


Figure 6.3: Comparing of JITting With and Without Inlining (Higher bars are better)

The JIT was run with the `-time-passes` option. So it was possible to relate the time spent in the JIT passes against the time the JIT was executing the application code. In Figure 6.3 one can see the toal and JIT exec time when executing `${APP}.llvm.bc` versus the time when running `${APP}.noinline.bc`. As already introduced, the difference between these two versions is that in the first version, the bytecode compiler and linker

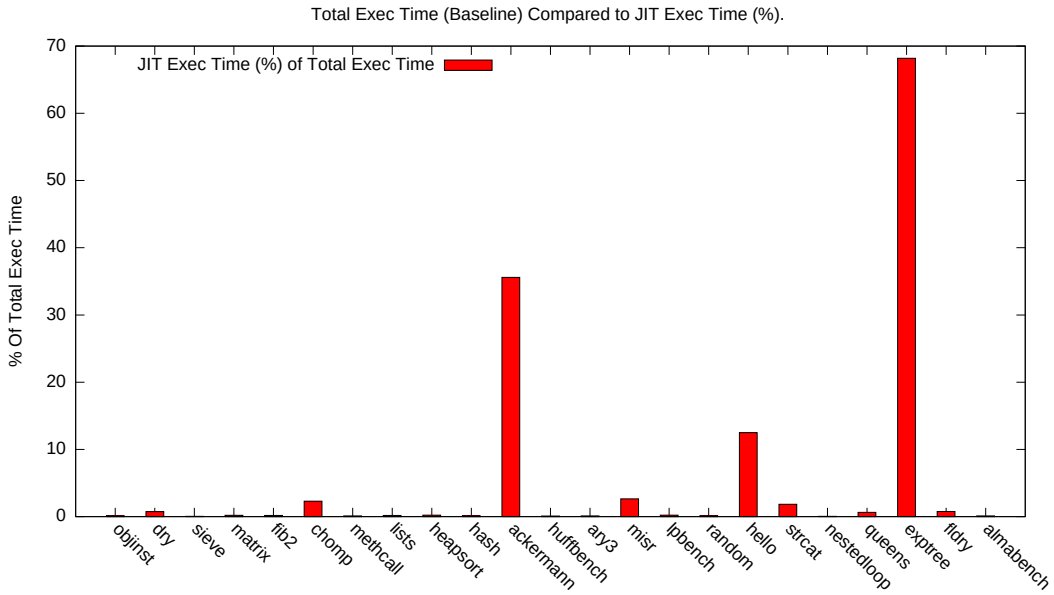


Figure 6.4: JIT Execution Time in % of Total Exec Time

heavily inlines functions, such that, for a single soucre application, the application often consists of a large inlined function, while in the other version the functions are not inlined at all. This resulted in two different bytecode files in use for testing. The inlined vs the non-inlined.

The red bar gives the baseline (100%) of execution time. The baseline is the execution time of the inlined version. Therefore the inlined version of the test application is only represented by a single red bar, since both the total exec time ($T_{Execution}$) as well as the JIT compile time ($T_{Translation}$) have both 100%. The green bar shows the speedup in total execution time ($T_{Execution}$) using the non-inlined LLVM version. The blue bar shows the speedup in JIT time ($T_{Translation}$), when using the non-inlined instead of the inlined version.

The general trend is that the two versions are almost as fast as the other. There was not much difference between using or omitting inlining. Exceptions are made by the *fib2* as well as the *ackermann* $T_{Translation}$ times. Here the non-inlined version of the bytecode compiles over two times faster than the inlined version.

As Figure 6.4 shows however, $T_{Translation}$ is much, much smaller than the $T_{Execution}$ of the application. This has to be kept in mind when looking at 6.3. Speedups in $T_{Translation}$ have very little influence on typical $T_{Exection}$ times. All the higher bars in 6.4 reflect short $T_{Exection}$ time. This is also true for the *exprtree* example, where $T_{Translation}$ was 70% of $T_{Execution}$. The overall execution time in this case was only $0.110s$, while the $T_{Translation}$ was $0.0775s$.

6.3.1.7 Results of Run-Time Measurements

In summary one can say, for the tested programs, with some exceptions, LLVM compiled code outperformed tests compiled with GCC. For most dynamically compiled tests $T_{Translation}$ is only a very small part of the overall $T_{Execution}$. If the translation time plays a bigger role in the overall execution time, it is often related to a very short execution time. For the tests $T_{Translation}$ never exceeded $100ms$ regardless if the bytecodes were inlined or not.

One interpretation is that the LLVM bytecodes opposed to high level VMs, like CLR or Java bytecodes are already very low level and targeted towards the machine code execution engine. So translating is a more light weight process and the code generation can be done efficiently.

6.3.1.8 Average Resident Set Size Memory

Another important issue in comparing the LLVM JIT vs machine code execution is the memory footprint of the applications. For measuring the memory footprint of an application the so called resident set size (RSS) was used.

RSS is resident memory size of the process. This is the amount of memory the operating system is not allowed to swap out. The resident memory size of a process is the code and data used most frequently. These are memory resident. Unfortunately there was no simple tool, like the `time(2)` utility available for measuring the memory in a batch fashion under the testing platform. In order to measure the average resident memory consumed by a process the author created a tool called `memmeter`. `memmeter` is used as shown below:

```
$ sudo memmeter /opt/llvm-1.9/bin/lli myapp.bc
average rss size kb: 3132
```

In the listing above, `memmeter` will execute the `lli` application with the argument `myapp.bc`. It does this by creating a child process which executes the passed application. The parent application waits for the child application to finish its work and periodically (every 100 ms) wakes up to calculate the resident set size of the child application.

The obtained average size is printed to `stderr` when the child application has finished executing. Figure 6.5 shows the percent in resident set size of the test applications during execution. The baseline resident size form the applications when compiled with the GCC suite. One can see that there is no difference between the memory usage of applications compiled using the GCC compiler (red bar) and the LLVM ahead of time compiler (green bar). The graph shows varying results for consumptions of the just-in-time-compiled applications (blue and violet bars).

In fact the just-in-time-compiled applications always consumes relatively more memory if the application itself does not consume much resident memory. This trend is comparable to the overall execution time versus the translation time. This stems from the fact that the just-in-time compiler has a certain fix cost. If the application itself consumes more

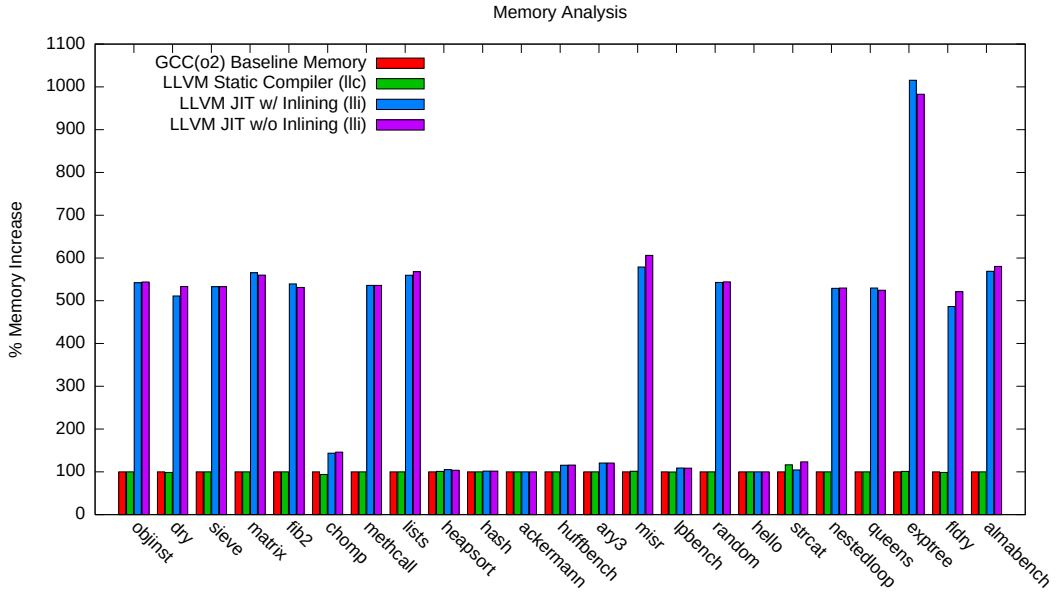


Figure 6.5: Resident Set Size of the Applications (Smaller bars are better)

time and memory the fix costs of the just-in-time environment are negligible. On the other hand if the application is very small both in terms of time and space the relative costs of the just-in-time environment are high.

The tests have also shown clearly that at minimum a machine code executable application needs around $600KB$, whereas the JIT processes at least needs around $3,300KB$ of real memory.

6.3.1.9 Conclusion of LLVM Run Time Infrastructure

In the tested application suites, the LLVM JIT has shown comparable performance to that of the ahead of time compiled executables. It could be also shown that the JIT has some fix costs both in time and space. The maximum translation time overhead in the tests were around $100ms$. On the other hand the memory overhead of the JIT were around $2,500KB$.

6.3.2 Evaluation of the Change Framework

The change framework builds upon the LLVM JIT to be able to consume changes, dynamically transform the LLVM representation, recompile the machine code, and relink the application so that the changed code gets executed. Additionally the changes have to be logged, so that, when a change is not needed any more, the old representation can be restored. The evaluation of the change framework builds on top of the performance information collected by the evaluation of the LLVM infrastructure. The current implementation status of the change framework is a proof of concept prototype. Evaluation will take the maturity of the implementation into account.

The evaluation of the change framework will be split into two parts: The parts of the change framework that provide the core functionality and therefore is present as soon as a program is run within the change framework are evaluated first. The other evaluation consists of that parts that are added when changes are applied and unapplied.

6.3.2.1 Core Run-Time Overhead

At the basis the change framework adds the following overhead:

- Instrument the code at load-time to add *recompilation checkpoints*.
- Change detection routines at so called *recompilation checkpoints* (See Section 5.5).
- Provider library.
- Core data structures for keeping track of changes.

Additionally, depending on its use, the change framework imposes restriction on the LLVM representation: For instance aggressive inlining is problematic if one wants to have a high level program representation. Insertion of recompilation checkpoints in the code is done by load time bytecode transformation. When the bytecode representation gets loaded a bytecode transformer will instrument function call instructions, which, in LLVM are the: `call`, and `invoke` instructions.

Currently a recompilation checkpoint is implemented as shown in Listing 6.1.

```
extern bool need_to_recompile;

void function1()
{
  ...
}

void function2()
{
  ...
  if (need_to_recompile!=0)
  {
    need_to_recompile = 0;
    recompile_handler_func();
  }
  function1();
  ...
}
```

Listing 6.1: Simple Example of Recompilation Checkpoints

Listing 6.1 shows that before a function is called, the following has to be performed:

- Load a global variable into a register.
- Compare the register to 0.
- Perform a conditional jump.

On the test machine, the check for recompilation would take something around 5 nanoseconds. For reference Figure 6.6 shows the time scale on a 1 GHz processor. As the re-

compilation checkpoint is instrumented into the application at every call-site, care must be taken, since even if the overhead is minimal it is nevertheless frequently executed.

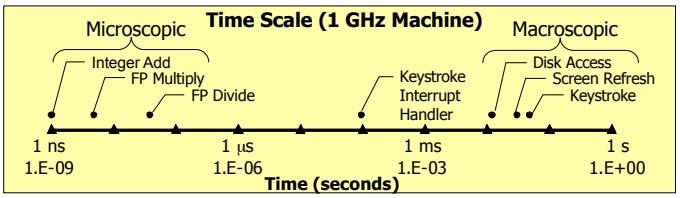


Figure 6.6: Time Scales, Lecture Notest for CSC 469H1F, [Bro06]

The execution time overhead, if no changes are applied, currently is negligible for most applications. The memory requirements applications that get just-in-time compiled are higher than the memory requirements for machine code executables. The change framework in addition has to store the following run-time data: For instance providers have to be registered, a change log has to be kept over registered changes, connected monitor processes, and IO connections have to be maintained. At the current state of implementation these lists are often hash tables, for providing fast access to the data, by some keys.

Change undo information has to be kept in memory as long as the changes are applied. After unapplication of a change all the run-time overhead should be recycled, such that other applications are able to register. In case no changes are applied/unapplied, the main run-time overhead is only minimal in addition to the memory requirements of the JIT.

6.3.2.2 Change Overhead

As documented in Subsection 5.2.2, when a change gets registered, the change provider transforms the regions of the LLVM representation, where change points are identified. This yields a recompilation of the LLVM representation and execution of the newly compiled functions.

This means no generic handlers have to be called, and the changed is added as if the application logic would be modified manually and recompiled. The performance impact of such a generic system is hard to measure. If the added change is implemented poorly, the application performance will go down dramatically. The change framework can be extended on two axes: For performance sensitive operations, providers and so called implicit values can be plugged into the virtual machine. For writing changes at certain change points, change scripts can be written.

As an example of the performance impact we consider a monitoring application. The output of measurements for instance has to be transported back to the monitor process. If a change is performed by a specific monitor application, the output has to be seen on that monitor's output, and not on the applications console.

```
change functionProvider::OnEnter
//
{
  io->printf( "==> Entering function '%s'\n", functionProvider->
    functionName );
}
```

Listing 6.2: Change to Print Out the Function Name.

Listing 6.2 shows a sample change script. The scripts prints every function that gets called onto a monitor process' output device. The `io->printf` is different than a local `printf`. In the above case a message gets transmitted to the monitoring application over the change framework's communication framework. For more details refer to Subsection 5.6.2.

In essence the simple call to `io->printf(...)` imposes a lot of run-time overhead. Care must be taken to implement this in an efficient manner. In a production system all *oneway messages*, such as `printf`, could be implemented as non-blocking calls. Despite the fact that implementations like that of implicit (or predefined) values are in principal beyond the scope of the change framework core, they get called at run-time and therefore have a noticeable impact on the overall performance. Well known values, such as for instance the `io` and `thread` values, should be implemented as low level and efficient as possible. [CSL04] implements a very efficient ring buffer in order to minimize run-time overhead.

6.4 Summary

This chapter evaluated the LLVM infrastructure as well as the change framework and the current prototype implementation. The performance data shows, that the LLVM JIT is in most cases as fast as an ordinary machine code executable, compiled with GCC using medium compiler optimizations (`-O2`), which means the GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff, and does not do loop unrolling or function inlining.

Also since the LLVM intermediate representation is already low level, JITing is negligible for the execution time on the test cases performed. One drawback of dynamic compiled system level code is the amount of memory needed by the user space process. The fix cost overhead of the JITing the LLVM code is about $2.5MB$. The more memory the application itself requires, the less important are the dynamic execution engines' memory requirements.

The primary goal of the implemented prototype was to produce a working prototype with a focus on producing high performance code. For most typical applications the primary overhead of the current prototype is not in the execution time, but in the memory consumption. Detecting whether a function needs recompilation takes about five to ten nanoseconds on our test machine. On the other hand there are a lot information to keep track of, which results in increasing memory consumption. Typical applications, including server systems and interactive applications could be run under the change framework on a modern system without noticeable performance impediments.

Chapter 7

Related Work

7.1 Overview

In this chapter some influencing and related work are introduced. There are two kinds of instrumentation systems in use. On the one hand there are binary code patching systems and on the other hand there are virtual machine based bytecode transformation systems. One can distinguish application level and operating system level transformation systems.

Binary code patching systems work by using debug symbols of a program to disassemble code into a higher representation. This information can be used to guide the system at which points the code should be patched. We will show this system based on the Dyninst API [Dyn06]. The Dyninst API is an application level transformation system. Solaris DTrace [CSL04] is an operating system level dynamic instrumentation framework. It also has the ability to patch binary code. Here the difference is that the framework runs within the kernel level. Hence the framework has access to all applications and the monitor interacts not directly with the transformed application, but uses a special system call interface.

High level process virtual machines, such as the Microsoft CLI, or the Java virtual machine (JVM), dynamically execute an intermediate language. This intermediate language typically has richer semantic information about the program being executed. Instrumentation here is done by transforming the bytecodes before they are loaded into the virtual machine. Dynamic instrumentation can be obtained by so called bytecode replacement at run-time (also referred to as hot swapping). Java has a feature called class redefinition. This works by telling the VM to redefine the bytecodes for the methods of a specific class. The Java Virtual Machine Tool Interface (JVMTI) is introduced as an example.

7.2 Dyninst API

The Dyninst [Dyn06] API was first described in [HMC94]. Dyninst is the infrastructure behind the Paradyn [Par06] project for parallel performance tools. Instead of working by transforming a V-ISA bytecode, Dyninst directly patches the machine code image of a running application. This has the advantage of not having a specific prepared

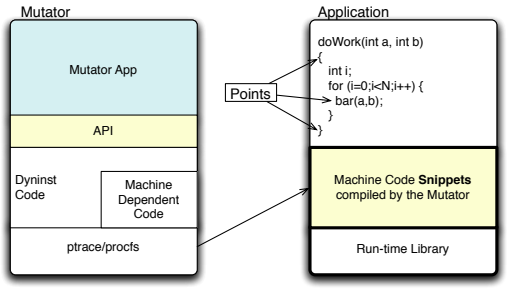


Figure 7.1: The Dyninst API

application process. Any installed application software with debug information can be instrumented.

As the change framework, the Dyninst API framework consists of two processes. There is a mutator process and an application process. Figure 7.1 shows the Dyninst API abstractions defined in [BH00]. The applications of the Dyninst API are closely related to the one of the change framework:

- Debugging
- Performance monitoring
- Altering application semantics
- Manipulate application data

Though the Dyninst API patches native machine code, the internal representation is machine independent. Every target machine reader provides almost the same abstract intermediate representation. Thus a mutator application program, as in Figure 7.1, can theoretically be used on all target machines simply by recompiling the application. So in the Dyninst API the machine code is read and an intermediate code representation is built. This representation is then edited by a mutator application. The altered intermediate representation is then compiled to native code again.

The steps of code patching are:

- Attach to a running program.
- Create a bit of machine code.
- Insert it into the program.
- The program being modified is able to continue execution, it does not have to be re-compiled, re-linked, or re-started.

The Dyninst API abstracts a program and its state while executing. Like in the change framework one primary abstraction is the *point*, another one is the *snippet*. Snippets are like changes in the change framework. Snippets represent executable code to be inserted into a program at a point. Furthermore the API provides abstractions for process images and threads of execution.

To refer to types in the running application a simple type system is supported. The simple type system provides support for integers, strings, and floating point values. In addition, aggregated types including arrays and structures are also supported. Out of

the box there is no way to create new types using the Dyninst API. Change framework type system is that of the LLVM architecture. Since the change framework transforms LLVM bytecode, it supports the LLVM type system. The current framework does not support the creation of custom types.

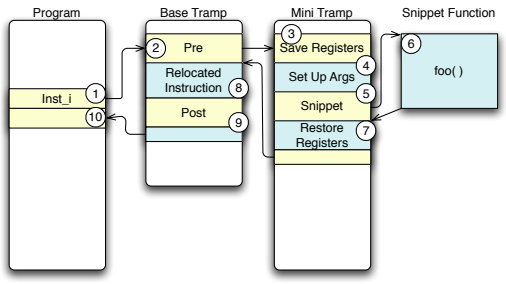


Figure 7.2: Inserting Code into a Running Program

Figure 7.2 shows the insertion of a snippet into a running application. The function `foo` is executed before the execution of an instruction $inst_i$. According to [BH00] the most difficult part of inserting instrumentation is carefully modifying the original code to branch into the newly generated code. As can be seen in Figure 7.2, the Dyninst API use short sections of code called *trampolines*. Trampolines are the means to get from the point where the instrumentation code should be inserted to the newly generated code.

So called mini trampolines are called from the base trampolines. Mini trampolines save the appropriate machine state and contain the code for a single code snippet. In Figure 7.2 the code is executed before the instruction. If one wants to execute the change after the instruction, the mini trampoline can be registered with in the Post list. Chaining of code snippets can be done by chaining mini trampolines.

7.3 Solaris DTrace

Solaris DTrace [CSL04] aims at dynamic instrumentation of production systems. Both user level and kernel level software can be instrumented in a safe and dynamic fashion.

The change framework is similar to DTrace with regard to safe and dynamic instrumentation of running applications. DTrace however is a kernel level instrumentation application. All instrumentation, probe processing, and buffering resides in the kernel space. Application processes running in user level can become DTrace consumers via the DTrace library. The DTrace library initiates communication with the in kernel DTrace subsystem. DTrace also has the concept of providers and its own instrumentation language. The framework itself performs no instrumentation. Providers have the form of loadable kernel modules and communicate with the DTrace core module via a well-defined API.

The DTrace framework instructs providers to determine potential instrumentation points. For every point of instrumentation, providers call back into the DTrace framework to create a `probe`. Hence a probe identifies a provider specific instrumentation

point. A probe is composed of the defining provider name, the kernel module name, the function name, and a probe name. The DTrace provider architecture influenced the design of the change framework architecture.

7.4 Java Virtual Machine Tool Interface

The Java Virtual Machine Tool Interface (JVMTI) [JVM06] is part of the Java Standard Edition since version 1.5. JVMTI provides several interfaces to hook into the Java virtual machine in a standardized way. JVMTI clients are so called agents. These agents use interfaces provided by the JVMTI. The agents have to be registered with the JVM. The JVM implements the JVMTI and hence provide the interface to the agents.

One can write JVMTI agents that gather information and extend the JVM behavior. JVMTI consists of a Java API as well as a native API. The change framework supports extensions too. Opposed to JVMTI, the change framework allows one to extend the functionality by writing a provider. In JVMTI the interface to the Java virutal machine (JVM$^{\mathrm{TM}}$) is fixed, such that one is restricted to the offered interface.

Since JVM$^{\mathrm{TM}}$1.5 Java too is able to perform hot code replacement, called class redefinition. This means as long as a Java class has no schema changes, the class can change its behavior (methods) without stopping and reloading an application or a class. As the JVMTI, the change framework is able to change code at run-time. In the change framework changes can additionally be unapplied too. This is done by keeping a change log on a LLVM instruction granularity. If a change is unregistered the bytecode again is transformed and recompiled. This is not directly supported by the JVMTI. Furthermore the granularity of redefinition in the JVMTI is that of a class. This makes keeping undo logs more expensive. One has to keep the old version of the class bytes available, so that one can revert to the old class files.

The main difference between the JVMTI and the change framework is that the primary way of instrumentation in Java is load-time instrumentation. For instance one is not allowed to perform so called schema changes to classes, that means to add or remove methods or fields or to change the visibility of classes. Except for bootstrapping purposes the change framework does not perform load-time transformation. In the change framework the unit of replacement can be defined by providers.

Chapter 8

Summary

The main objective of this thesis is the description and specification of the change framework. As detailed in Chapter 5 the change framework is a system for dynamic program instrumentation and analysis based on the Low Level Virtual Machine (LLVM) architecture. This chapter will be organized as follows. First a conclusion is given, then several sections discuss future work. Finally the last section shows some of the current limitations of the change framework prototype.

8.1 Conclusions

The change framework offers the ability to dynamically change the behavior of a program while being able to later revert the changes again. While there are many approaches for this problem, the one chosen by this project was to dynamically compile an application on a just-in-time basis. The representation and infrastructure is the Low Level Virtual Machine (LLVM) project. Part of this project is a low level program representation that makes it possible to represent C and C++ system level code efficiently, while remaining more details of the program structure, such as high-level data types.

In order for the system to remain flexible and to separate the low level details of the virtual machine from the authors of change scripts, the change language was defined. The change language is itself compiled to LLVM, which makes the transformation process very straightforward. The change framework by itself is very flexible. The main usage scenario outlines in this thesis was interactive program analysis and program understanding. However this is only one aspect of dynamic program transformation, and the change framework is open to be used for different scenarios.

In the change framework system, a monitor processes compiles a change script, authored by a change author. The change script is sent to the application process, that in turn applies the changes and sends information back to the monitor process. The monitor process can later revert any changes performed. This enables interactive usage patterns where one first applies more generic instrumentation and later as the problem understanding grows, narrows the instrumentation towards the problem areas. In Chapter 6 the system was evaluated. One of the principal outcomes was that the run-time overhead of the dynamic compiler is negligible. The biggest obstacle for dynamic compiling programs is the additional memory requirements.

Nevertheless it was shown that the implementation of the change framework is feasible. Especially long running applications would benefit from systems like the change framework. Problems could be isolated without having to stop the application as a whole.

8.2 Future Work Overview

The next sections discuss future work of the change framework. For clarity purposes the issues will be split into:

- *LLVM enhancements:* While the LLVM serves as a great platform for the change framework, the more meta information available at change application time, the easier change authoring gets. That means the more closely the intermediate language resembles the original programs intent, the better it can be analyzed. This is somewhat contradictive, since the LLVM tries to be a common program representation.
- *Language enhancements:* The section discusses enhancing the change language. Concepts such as provider inheritance, type variables, support for pattern matching are discussed.
- *Change framework enhancements:* This section gives an overview of future work on the change framework. Example enhancements are introduced and possible implementations are discussed. Examples are: Separate program transformation from information providers, provide an abstract language for defining providers, allow the system to be used with ahead of time compiled applications, and provide infrastructure for alternative backends other than LLVM.

8.3 LLVM Enhancements

While LLVM serves as a good platform for the change framework, there are some areas which could be improved to provide more information for change points. In the following subsection one specific issue will be discussed. The `struct` type is used to represent C `struct`s or Pascal `record` types. When for instance a C code snippet like the one in 8.1 is translated to LLVM, the symbolic information is completely lost.

```
struct Person {
  char *firstname;
  char *lastname;
  int   age;
};
```

Listing 8.1: Example C Structure

The translated LLVM struct is shown in 8.2. As can be seen the symbolic information of fields is lost.

```
%struct.Person = type { sbyte*, sbyte*, int }
```

Listing 8.2: Example LLVM Structure

```
void person_set_name(Person* person, char *firstname, char *lastname )
{
  /* assert that the arguments are fitting ... */
  person->firstname = strdup(firstname);
  person->lastname  = strdup(lastname);
}
```

Listing 8.3: Using the Structure in C

Listing 8.4 shows the translation of the code of Listing 8.3 from C to the equivalent LLVM representation. One can see that in the LLVM version, there is no symbolic information of fields. The `getelementptr` instruction is used to obtain a pointer to the fields. The pointer to a field of the struct will be obtained using its slot number in the struct type. Since the type information of the fields is preserved the pointer can be calculated by using the slot number of the field in question.

```
%struct.Person = type { sbyte*, sbyte*, int }
void %person_set_name(%struct.Person* %person, sbyte* %firstname,
    sbyte* %lastname)
{
  %f = call sbyte* strdup(%firstname)
  %tmp = getelementptr %struct.Person* %person, int 0, uint 0 ; <sbyte
    **>
  store sbyte* %f, sbyte** %tmp

  %l = call sbyte* strdup(%lastname)
  %tmp1 = getelementptr %struct.Person* %person, int 0, uint 1 ; <
    sbyte**>
  store sbyte* %l, sbyte** %tmp1
}
```

Listing 8.4: Using the Structure in LLVM

As can be seen in Listing 8.4, the symbolic information of filed names is completely lost in the LLVM representation. What is technically correct for low level representation can be problematic for frameworks such as the change framework.

```
basicblock provider FieldStoreProvider
{
  points {Before, After};
  string structType;
  string fieldName;
};
```

Listing 8.5: Provider exporting stores to fields

This low-level property of the LLVM makes it difficult to write change predicates that filter for field names. The Listing 8.6 shows a change snippet, that uses the provider from Listing 8.5. The change wants to intercept all writes to a field called ”lastname”. As can be seen this is would be an interesting question to ask about a program.

```
change FieldStoreProvider::Before
/ FieldStoreProvider->fieldName == "lastname" /
{
  io->printf("Before␣storing␣a␣field␣%s,␣on␣struct␣%s",
             FieldStoreProvider->fieldName,
             FieldStoreProvider->structType );
```

```
}
```

Listing 8.6: Change Script That Uses the Field Store Provider

Without using additional meta information, for instance debug information, the code above could not be realized. The lack of symbolic information in this case is problematic. A possible enhancement of the LLVM would be to annotate the LLVM struct type to include symbolic information. An augmented struct type is shown in Listing 8.7.

```
%struct.Person = type
  {
    ("firstname",sbyte*),
    ("lastname",sbyte*),
    ("age",int)
  }
```

Listing 8.7: Enhanced LLVM Struct Types

With a type representation as above, the provider in Listing 8.5 could be accomplished. The `getelementptr` identifies the field slot. The provider could take the struct type and look for a given symbolic name. If the struct contains the name, the provider calculates the slot offset from the name. Then it instruments all the getelementptr instructions, whose struct type is the struct observed, and the slot is the calculated.

8.4 Language Enhancements

The change language, discussed in Section 5.4 is used to create application changes. The change language is based on the C language. It adds first class language support for change constructs, such as:

- change providers,
- change predicates and change body, and
- implicit (predefined) values.

The change language is lacking some advanced features, which are important in order to ease authoring of change scripts. Some of the improvement are:

- Provider inheritance
- Type variables
- Pattern matching support
- Separate provider name and exported variables

8.4.1 Provider Inheritance

Listing 8.8 shows a sample provider on an instruction granularity representing a generic instruction:

```
instruction provider InstProvider
{
  points {Before, After};
```

```
  string name;
  string type;
};
```

Listing 8.8: `InstProvider` - A Generic Instruction Provider

A refined provider based on the `InstProvider`, for instance for a special instruction class, would be implemented by copying the `InstProvider` and adding the new functionality. The derived provider shares no semantic relationship with the `InstProvider`. Listing 8.9 shows a `MallocProvdier`, which operates on instruction granularity and represents the `malloc` instruction.

```
instruction provider MallocProvider
{
  points {Before, After};
  string name;
  string type;
  string allocatedType;
};
```

Listing 8.9: `MallocProvider` Refining `InstProvider`

If the `InstProvider` is changed, one has to keep track of the changes in all of the manually refined providers. An improvement would be to derive from, refine, or sub class an existing provider.

Listing 8.10 shows the provider from Listing 8.9 with provider inheritance. Only the `allocatedType` was defined in the `MallocProvider`, the other properties are inherited from the parent provider `InstProvider`.

```
instruction provider InstProvider
{
  points {Before, After};
  string name;
  string type;
};

instruction provider MallocProvider refines InstProvider
{
  string allocatedType;
};
```

Listing 8.10: Provider Showing Inheritance

8.4.2 Type Expressions and Variables

Currently types are encoded by a string representation. This is sufficient for primitive types, such as `int`, `float`. When dealing with compound types, such as `structs`, arrays, or even pointers, string representations become tedious. Some of the problems are: A single type can have multiple valid string representations, the coding of compound types is not unique. Type matching has to be done using substring searches. The compiler is not able to understand the types, and therefore is not able to support the author.

In the C language, a type is used in declarations, but normally a type can not be stored in a variable and can be queried explicitly. Languages with support for introspection, like CLR, or Java, have a way to dynamically access the type of a class, field, or method. The concept introduced here are type variables, and type expressions. This means that one is able to build type values which can be stored in special variables, called type variables. These variables represent the meta information of the type. Additionally type expression can be used inside statements.

Listing 8.11 shows an example C program, Listing 8.13 gives an overview of a sample change language fragment. In this fragment a change predicate is used to query for a special kind of function based on the stringified type information exported by a provider shown in Listing 8.12.

```
int rect_square(int length, int width)
{
  return length * width;
}
```

Listing 8.11: Example Program

```
function provider functionProvider
{
  points { OnEnter, OnLeave };
  int     id;
  string  functionName;
  int     argCount;
  bool    isVarArg;
  string    signature;
  string    returnType;
  string[]  argumentTypes;
}
```

Listing 8.12: Example Provider

```
functionProvider::OnEnter
/ functionProvider->argCount == 2  &&
  functionProvider->returnType == "int" &&
  functionProvider->argumentTypes[0] == "int" &&
  functionProvider->argumentTypes[1] == "int"
/
{
  io->printf("Entering␣function␣'int␣%s(int,int)'\n", functionProvider
    ->functionName);
}
```

Listing 8.13: Changes Using Stringified Types

Listing 8.13 shows the current change language. In this listing functions of signature `int (int,int)` are matched and printed. Listing 8.16 shows the same listing with type variables. A type variable is a variable of type `type`. A type expression is constructed using a type constructor. A type constructor starts with `type`. Listing 8.14 gives some examples of type expressions bound to type variables.

```
type intType    = type int ;
type structType = type { int a; int b; int c; } ;
```

```
type funcType    = type int (int a, int b );
```

Listing 8.14: Sample Type Expressions

Using type variables and type expression the code showed in Listing 8.11 can be transformed to the concise form shown in Listing 8.15.

```
function provider functionProvider
{
  points { OnEnter, OnLeave };
  int     id;
  string functionName;
  type    funcType;
}
```

Listing 8.15: Example Provider

As can be seen in Listing 8.15, all the string type helper fields are replaced by a single field called `funcType` of type `type`.

Respectively the code in Listing 8.13 could be shortened to that of Listing 8.16.

```
functionProvider::OnEnter
/ functionProvider->funcType == type int (int, int) /
{
  io->printf("Entering␣function␣'int␣%s(int,int)'\n",
       functionProvider->functionName );
}
```

Listing 8.16: Changes Using Type Variables

Type variables would represent types as first class values in the change language. A type variable is a uniform representation of a type, stored as an abstract syntax tree.

8.4.3 Type Expression Pattern Matching

With type expressions and type variables the change language gets more concise. Often one is not interested in an exact type, but in a set of types. This subsection shortly discusses adding pattern matching to type expression. This would allow to write even more concise change scripts. Pattern matching is added to type expressions by allowing concrete type expression and expression patterns. Expression patterns are distinguished form type expressions by containing place holder elements. Pattern matching could be performed at run-time, by comparing the concrete type and the type pattern. Since patterns are static in nature, a more efficient way would be to compile the pattern into primitive match operations, that can be inlined. Regardless of the kind of matching, the implementation has to match a type T against a pattern P and return *true* if T matches P, $false$ otherwise.

$match : (T \times P) \rightarrow \{true, false\}$

Based on the change language grammar, introduced in Listing 5.4, Listing 8.17 shows a type expression grammar in EBNF. As shown in the listing `_` is the pattern matching place holder. Such a place holder can represent any `TypeExpr`.

```
AnyType       = '_' .
SkipTypes     = '...' .
TypeExpr      = 'type' InnerTypeExpr.
InnerTypeExpr = AnyType | SkipTypes | TypeName | TypeEl | TypePattern
    | TypeFunc | TypeStruct .
TypeFunc      = InnerTypeEpxr '(' {InnerTypeExpr} ')'.
TypeStruct    = '{' { InnerTypeExpr [ident] ';' } '}'
TypePattern   = '/' TypeExpr '/'.

EqualityExpr     = PatternMatchExpr { '==' PatternMatchExpr }.
PatternMatchExpr = RelationalExpr { 'matches' TypePattern }.
...
```

Listing 8.17: Augmented Change Language Grammar

An example type declaration is shown in Listing 8.18.

```
type PointType = type { int x; int y ; };
```

Listing 8.18: An Example Type Declaration

Listing 8.19 shows the usage of pattern matching expression.

```
bool matchesPoint = PointType matches / type { _ ; int ; } /
```

Listing 8.19: An Example Pattern Match

In Listing 8.19 above, the variable `matchesPoint` is true for the concrete type `PointType`. More precisely the pattern `/ _; int ; /` matches any `struct` with two fields. The first field can be of any type, the second field has to be of type `int`. Listing 8.20 shows pattern matching in the change language.

```
functionProvider::OnEnter
/ functionProvider->funcType matches / type _ (int, ..., float*) / /
{
  io->printf("Entering␣function␣'int␣%s(int,...,␣float*)'\n",
      functionProvider->functionName);
}
```

Listing 8.20: Change Using Type Patter Matching

In the above listing, the pattern `_ (int, ..., foat*)` matches any function type, whose first argument is of type `int`, and whose last argument is of type pointer to `float`. Further the function has to have at least 2 parameters, but can have more. The return type can be any type including `void`.

8.5 Framework Enhancements

The change framework consists of

- a *virtual execution engine* that dynamically translates a virtual ISA into a native ISA,
- a *change manager* as well as a set of *providers*,

- a *monitor process*, that sends and monitors changes, and
- a *change* written in the change language.

Among many other enhancements, the following future work would be useful:

- Separate program transformation from information providers.
- Provide an abstract language for defining providers.
- Allow the system to be used with ahead of time compiled applications.
- Alternative backends other than LLVM.

8.5.1 Separate Program Transformation and Information Providers

Currently a provider is implemented by a set of C++ classes as well as global registration code. After a provider is loaded into the application process, it is accessible from within the change language. Provider declarations represent providers in the change language. Listing 8.21 shows such a provider declaration.

```
function provider functionProvider
{
  points { OnEnter, OnLeave };
  int      id;
  string   functionName;
  int      argCount;
  bool     isVarArg;
  string   signature;
  string   returnType;
  string[] argumentTypes;
}
```

Listing 8.21: Example Function Provider

The provider above declares that it works on a function scope and identifies individual functions. Especially the provider identifies the points `OnEnter`, and `OnLeave`. `OnEnter` identifies entering the function, `OnLeave` identifies points in the function where the function is exited. The information is collected by the provider, it can be used inside change predicates and change statements. As explained in Chapter 5 every change uses a change provider. Change providers are in one of the following granularity: *module*, *function*, *basicblock*, and *instruction*. Changes that use a provider of a granularity g, automatically have access to all information exported by providers of a coarser granularity. In case a provider is used that works at an instruction scope, all information from basic block providers or function providers are accessible within the change fragment.

A future enhancement would be to separate the process of transformation and the process of providing context information more clearly. The change point contains the scope information implicitly. For instance in the current version a change point reference has to use the transforming provider.

```
/* Current usage of change points */
change functionProvider::OnEnter
/ functionProvider->functionName == "main" /
{
  /* do something */
}
```

```
/* Future enhancement */
change
/ function::Enter && functionProvider->functionName == "main" /
{
  /* do something */
}
```

Listing 8.22: Separate Change Points and Change Providers

Listing 8.22 shows the difference between the current model and the separation of the providers and the change point. The scope of the change and the transformation of the IR would be guided by the change point. Providers export information to the change framework. Listing 8.23 shows the new syntax for defining change points as well as information providers. Listing 8.24 shows sample changes, that uses the new syntax. As can bee seen the changes do not use change identifier anymore. Instead the change point is part of the predicate and determines where the change should get applied.

```
/* Sample points */
function point Enter ;
function point Leave ;
function point EnterLoop;
function point LeaveLoop;
function point LoopHeader;
instruction point Before;
instruction point After;

/* Sample providers */
function provider functionProvider {
   string functionName;
   /* ... */
};
```

Listing 8.23: Change Point Syntax

```
change
/ function::Enter && functionProvider->functionName == 'main' /
{ /* ... */ }

change
/ function::BeforeLoop &&
  functionProvider->functionName == 'do_work_a' /
{ /* ... */ }

change
/ instruction::Before && ... /
{ /* ... */ }
```

Listing 8.24: Usage of New Change Points

Listing 8.26 shows a change point identification in Java like pseudo code. An instruction changepoint occurs at any single instruction, a change point simply returns true or false for any instruction that is given as an input. For instance the `FunctionOnEnterPoint` returns true for instructions that are not a special `phi` or `alloca` instruction and are in the first basic block of the function. Since one is able to navigate the basic block and the function from the instruction object, the instruction is sufficient as an argument.

```
enum Match {
  NO_MATCH,
  BEFORE,
  AFTER
};
enum Granularity {
  MODULE,
  FUNCTION,
  BASIC_BLOCK,
  INSTRUCTION
};

interface InstructionChangePoint
{
  String getName();
  Match identify(Instruction inst);
  Granularity getGranularity();
}
```

Listing 8.25: Interface for Abstracting the Changepoint

```
class FunctionOnEnterPoint implements InstructionChangePoint
{

  /**
   * @returns the name of the change point.
   */
  String getName() { return "Enter" }

  /**
   * @returns true if inst is a function enter point
   *          false otherwise
   */
  Match identify(Instruction inst)
  {
    BasicBlock firstBB = inst.getFunction().getFirstBasicBlock();
    if (firstBB != inst.getBasicBlock()) return Match.NO_MATCH;

    return (Function.isEnterPoint(firstBB, inst) ?
            Match.BEFORE : Match.NO_MATCH);
  }

  /**
   * @returns the granularity of the change point.
   */
  Granularity getGranularity() { return Granularity.FUNCTION };
};
```

Listing 8.26: A Change Point Library

Such a changepoint implementation as shown in listing 8.26 could be used in the change language as follows:

```
function point Enter;

change / function::Enter / { /* */ }
```

This separation between the identification of instructions which are a change points and the information exporting providers, makes it easier to implement providers as well as enables a generic weaver that could insert the change into the code transparent of the provider implementation.

8.5.2 Abstract Provider Language

Most of the complexity of a provider lies in matching the right instruction to transform. In the current model writing a provider imposes one to know the LLVM libraries. This makes authoring providers harder than it should be. On the other hand providers are the only way how changes are able to use new functionality. One way would be to provide a language that builds upon the change language and the new features discussed in Section 8.4. The provider language would make it possible to write providers with a comfortable domain specific language without much knowledge of the underlying virtual machine. The providers would have to be compiled to LLVM bytecode too, in order to be usable.

8.5.3 Ahead Of Time Compiled Applications

The change framework is not limited to JIT compilation. Figure 8.5.3 shows the change framework when running as dedicated service.

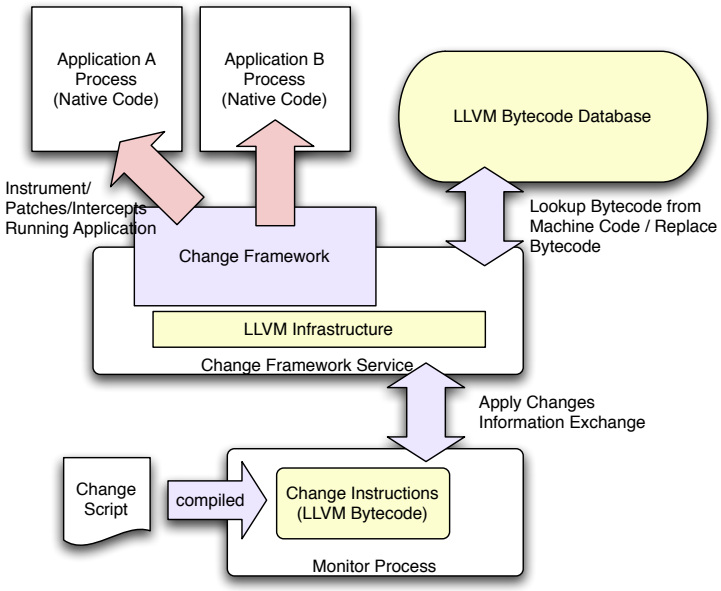


Figure 8.1: Change Framework as External Service

In this approach the applications are compiled to LLVM bytecode. Later the applications are instrumented, and ahead of time compiled to native code. This means no dynamic compilation is needed, if the application is not dynamically monitored using the change framework. If a monitor process is used to change a running application it contacts the change framework service that runs in a separated process. The change framework

service then uses a bytecode database in order to lookup the bytecode for the target application process.

All the dynamic compiler infrastructure operates in the dedicated change framework service process. If a change affects a function in an application process, the function is transformed. The changed bytecode is stored again in the bytecode database. The changed bytecodes are then compiled into native code using the LLVM infrastructure. Changing the application process is done by injecting the compiled function into the target application process and further by invalidating the old function. Technically there are many possibilities to replace the function within the native code application. One is by using toolkits such as [Par06].

The advantages of the external change framework service are that not each process under analysis has to run within the execution engine yet that if one maintains the bytecode database at installation time, any application that was ahead-of-time compiled to LLVM bytecodes is subject to dynamic instrumentation. Thus one does not need to decide at application start time, whether one wants to make use of the change framework.

8.6 Limitations of the Change Framework Prototype

This section will discuss some of the limitations of the current implementation of the change framework. As described in Chapter 6, the current status of the change framework is that of a prototype proof of concept. This section shortly discusses limitations based on:

- Change framework limitations: These are limitations in the application process.
- Change language compiler limitations: Limitations of the change language compiler.

8.6.1 Change Framework Limitations

- The LLVM version used by the change framework is limited to version 1.5.
- `ImplicitValues` are hard coded. There is currently no API that enables a pluggable interface for adding implicit values to the change language, such as `io`.
- Providers have to be declared in the change language as well as in C/C++ code.
- No standard provider attributes and naming schemes are documented.
- Checking for recompilation is done by checking for a shared global variable.
- The native code buffer is limited to 64 MB. If many changes are applied the buffer could overflow.
- Relinking a function is done by adding a direct jump into the old function in order to unconditionally jump to the new one. This results in a slow down of the call chain.
- Detection of a change is only done on a per function basis. On stack replacement [ES00] could drastically improve responsiveness.
- The monitor process does not provide an interactive shell for applying multiple changes.

8.6.2 Change Language Compiler Limitations

- The current change language parser lacks sophisticated error correction.
- The error codes outputted from the compiler are currently not internationalized.
- There is no way to dump the AST of the change language.
- An application binary interface (ABI) check is missing for the current compiler.
- There are no regression tests available for the change language.
- The compiler was only tested under GNU/Linux.

8.6.3 Summary

Not all limitations are listed here. While the current framework is not production ready the proof of concept was successful. Many of the limitations are out of the scope of this master thesis, for which only a limited implementation time frame was available.

Bibliography

[ALB+06] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Llva: A low-level virtual instruction set architecture, 2006.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[Bri] Misha Brukman Brian. Cs 497yyz project report: Llva-emu.

[Bro06] Angela Demke Brown. Lecture 5: Performance Evaluation. WWW resource, 2006. `http://www.cs.toronto.edu/~demke/469F.06/Lectures/Lecture5.pdf`.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Coy04] Comparinng Linux Compilers. WWW page, 2004. `http://www.coyotegulch.com/reviews/linux_compilers/index.html`.

[CP95] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *The First ACM SIGPLAN Workshop on Intermediate Representations*, San Francisco, CA, 1995.

[CSL04] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, San Francisco, 2004.

[Dmi01] M. Dmitriev. Safe class and data evolution in large and long-lived java[tm] applications, 2001.

[Dyn06] Dyninst API Website. WWW page, 2006. `http://www.dyninst.org`.

[ES00] Matthias Ernst and Daniel Schneider. Konzepte und Implementierungen moderner virtueller Maschinen. Master's thesis, 2000.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, 1994.

[Hoe94] U. Hoelzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD dissertation, 1994.

[JVM06] JVM Tool Interface. WWW page, 2006. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html`.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lif06] The Lifelong Code Optimization Project. WWW page, 2006. `http://www-faculty.cs.uiuc.edu/~vadve/lcoproject.html`.

[LLV06a] LLVM Bytecode Format. WWW page, 2006. `http://llvm.org/docs/BytecodeFormat.html`.

[LLV06b] Extending LLVM: Adding instructions, intrinsics, types, etc. WWW page, 2006. `http://llvm.org/docs/ExtendingLLVM.html`.

[LLV06c] LLVM Assembly Language Reference Manual. WWW page, 2006. `http://llvm.org/docs/LangRef.html`.

[LLV06d] Writing an LLVM Pass. WWW page, 2006. `http://llvm.org/docs/WritingAnLLVMPass.html`.

[LLV06e] LLVM Website. WWW page, 2006. `http://www.llvm.org`.

[Moe03] Hanspeter Moessenboeck. Unterlagen zur Vorlesung Übersetzerbau, 2003. `http://www.ssw.uni-linz.ac.at/Teaching/Lectures/UB/VL`.

[Muc98] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, repr. edition, 1998.

[Par06] Paradyn Parallel Performance Tools. WWW page, 2006. `http://www.paradyn.org`.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, June 2005.

[Wir77] Niklaus Wirth. What can we do about the unecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20:822–823, November 1977.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz Urfahr, am April 25, 2007

Jakob Praher