

Chapter 2

Generating, Compiling and Executing Bytecode Programs

Abstract The generation of CIL programs—input of ILDJIT—is an important step of the compilation process described inside this book. This chapter describes how to generate CIL code by using available tools, such as Mono and GCC4CLI. Moreover, compilation processes available in ILDJIT are described including both static and dynamic compilations.

Keywords CIL programs · Compile bytecode programs · Compilation framework · Just-in-time compilation · Dynamic look-ahead compilation · Dynamic compilation · Static compilation · Bytecode systems

Compilers consider programs written by using a source language, such as C, C++, Java, and so on, to generate their semantically equivalent representations by targeting a destination language, such as the machine code of a target platform (e.g. Intel x86). In order to start using them, examples of programs written by using their source language have to be provided.

ILDJIT [1] is able both to translate programs written in CIL [2] bytecode language and execute them on Intel x86 and ARM platforms. Therefore, in order to use this framework, we need to produce the input of ILDJIT: CIL programs.

CIL is a stack-based bytecode language with a rich set of metadata, which includes descriptions of data types. Even if it is possible to write CIL programs by using a normal text editor, such as `vi` or `emacs`, and encode them to its binary format described in the ECMA-335 [2] standard, the author of this book suggests to the reader to use available tools to generate them automatically by starting from more human readable programming languages like C [3], C++ [4], Java [5] or C# [6] to avoid possible headaches.

This chapter starts by describing how to generate CIL programs from C or C# programs and it continues by introducing the compilation process of ILDJIT, which leads to the generation of the target machine code eventually. Different compilation schemes are available to the user of ILDJIT, which can decide the set of compilation

steps to perform offline (i.e., static compilation) and the ones to perform when the program is running (i.e., dynamic compilation).

2.1 Generating the Bytecode

In order to describe how to generate CIL programs, we start writing one of the simplest program from the user of a programming language point of view, which is also one of the first milestone for a compiler developer: the famous Hello world program. After the introduction of its implementation in two different languages, C and C#, we generate their correspondent CIL representations to highlight the impact of the source language to the CIL program.

By using an editor like `vi`, we write the following program and we save it to the file called `hello_world.c`:

```
/* hello_world.c */
#include <stdio.h>
int main () {
    printf("Hello, world!\n");
    return 0;
}
```

To produce the CIL bytecode from C programs, we rely on the GCC based compiler called GCC4CLI [7]. In particular, in order to compile our C program, we run the following command:

```
$ cil32-gcc -o hello_world_c.cil hello_world.c
```

The result is a file called `hello_world_c.cil`, which is the CIL representation of our hello world program. This file can be used as input to ILDJIT, which first produces and then executes the correspondent machine code. In order to run CIL programs, the ILDJIT command `iljit` is used and its syntax is the following:

```
iljit ILDJIT_OPTIONS FILE_CIL ARGUMENTS_OF_CIL_PROGRAM
```

In our case, the hello world program has no parameter and we do not use ILDJIT options for its execution. Hence,

```
$ iljit hello_world_c.cil
Hello, world!
```

Notice that we cannot execute CIL programs directly on our underlying platform because it does not know how to interpret them.

As next example, we generate the CIL representation of a hello world program by using the C# programming language. By using our editor, we write the following file called `hello_world.cs`:

```
/* hello_world.cs */
using System;
public class HelloWorld {
public static int Main () {
    Console.WriteLine("Hello, world!\n");
    return 0;
}
}
```

In order to produce the CIL representation of our C# program, we rely on the Mono [8] compiler.

```
$ mcs -out:hello_world_cs.cil hello_world.cs
```

The result is a file called `hello_world_cs.cil`, which is the CIL representation of our C# hello world program. As before, the generated file is given as input to ILDJIT.

```
$ iljit hello_world_cs.cil
Hello, world!
```

Since from two different programming languages, C and C#, we generate two different CIL programs that produce the same output, one question could arise: what are the differences between these two CIL programs? To answer this question, we compare those two programs: `hello_world_c.cil` and `hello_world_cs.cil`. A deeper analysis on these files (following described) shows that they differ quite substantially. This difference exists for two reasons: first these files are produced starting from different programming languages, C and C#. The second reason is due to the different compiler used to produce them: GCC4CLI and Mono. Our analysis is based on the output of the tool `monodis` [8] available inside the Mono project. Following we report the important fraction of its output when it is applied to `hello_world_cs.cil`.

```
$ monodis ./hello_world_cs.cil
...
.method public static int32 Main () cil managed {
    ldstr "Hello, world!\n"
    call void class System.Console::Write(string)
    ldc.i4.0
    ret
}
...
```

The CIL is composed by one method, `Main`, which is the entry point of the program (i.e., the first method executed). Inside this method, there is a call to another one, `Write`, which belongs to an external CIL library called Base Class Library

(BCL), which is defined inside the ECMA-335 standard. The method `Write` prints to the terminal the string given as its input eventually.

Let us consider the CIL hello world program coming from the C language: `hello_world_cs.cil`. As before, we use `monodis` to analyze the file. The important fraction of its output is the following:

```
$ monodis ./hello_world_c.cil
...
.method public static int32 main () cil managed {
  ldsflda valuetype string_type hello_world_c::string 0
  call int32 libstd::puts(int8*)
  pop
  ldc.i4 0
  ret
}
...
```

We can notice two important differences comparing it to the previous case: the function called to print the string is different. Instead of calling `Write`, the method `main` calls `puts`, which belongs to an external CIL library produced by the GCC4CLI compiler, that maps calls to the standard C library to the ones available in BCL. We can assign this mismatch to the differences between the two programming languages used. The second difference is related to the string to print. This time, the mismatch is due to the fact that different compilers have been used to generate the CIL.

2.2 Static Compilation

ILDJIT provides a compilation scheme that exposes the benefits of the static compilation by exploiting the more and more predominant multicore technology. After describing what a static compilation scheme is, we introduce and motivate its specific implementation available inside ILDJIT.

Usually, static compilation refers to the compilation process where the correspondent machine code of a target platform is produced from a program written by using an high level language, such as C, C#, CIL, and so on. More generally, a static compilation is the process of translating a program, available in a source language, into a target language, and its storing inside a file system, which makes possible later executions of it without necessitating the same compilation process anymore. Notice that static compilations do not assume the knowledge of input data of programs given as input. An example of static compilation is the translation from C to CIL described in [Sect. 2.1](#).

As previously described, CIL programs rely on the standard library called BCL. This library is composed by several classes, which include methods; some of these

Fig. 2.1 Execution of a CIL program by using ILDJIT, which is composed by a compiler and a runtime system

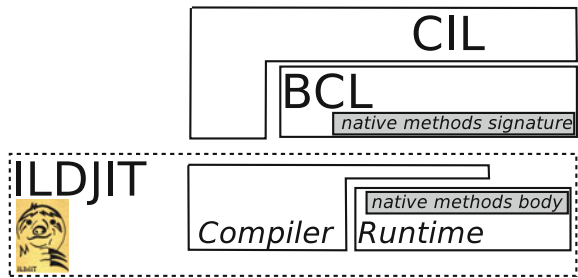
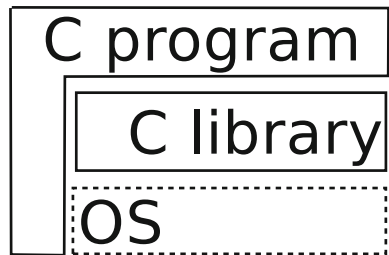


Fig. 2.2 Execution of a program written in the C language and compiled it statically



methods have their signatures described in CIL, but their bodies have to be provided by the runtime system—ILDJIT in our case. Most of these native methods are similar to the classic system calls of the underlying operating system; an example is `Platform.FileMethods.Open`, which opens a file whose name is specified as input by an object `String`. The relation between CIL programs, BCL and ILDJIT is shown in Fig. 2.1. The CIL program is translated to the machine code of the underlying platform before its execution by the compiler available inside ILDJIT. This transformation can be applied anytime (e.g., at static time or at runtime). The runtime system built inside ILDJIT is in charge of virtualize the underlying operating system by exposing the bodies of the aforementioned native methods like `Platform.FileMethods.Open`.

In order to better understand possible implementations of the static compilation scheme inside environments like CIL (similar discussion can be done for the Java environment), in the following, we describe the execution of programs written in a programming language where this scheme is usually applied: the C language.

Consider a C program for example: it is translated to machine code by a compiler like `gcc` to be later executed. The environment where these programs are executed is shown in Fig. 2.2.

The first difference between the execution of CIL and C programs is about the compiler: after C programs have been translated, the compiler does not play any role during their executions. In other words, the compiler is detached from the running produced code. On the other hand, at least CIL programs need a runtime, which is coupled with their executions; this fact suggests implementations of compilation schemes, which are usually applied to bytecode systems, where the compiler, which

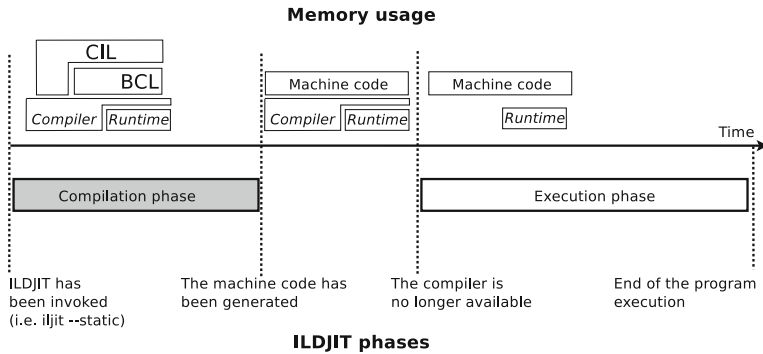


Fig. 2.3 Static compilation scheme implemented inside ILDJIT

is part of the virtual machine used, is kept in memory even after the compilation of the running bytecode program.

Another difference is related to the machine code generated: code generated from C programs interacts with the underlying operating system directly. On the other hand, code generated by the compiler built inside virtual machines, such as ILDJIT, interacts with the runtime, which can perform additional checks before redirecting the execution to the underlying operating system. This indirection can be used to improve security inside a system.

As programs that run in user space need an operating system, as CIL programs need a runtime system that provides the same kind of abstraction for the underlying platform. For this reason, ILDJIT implements the static compilation scheme in a slightly different way with respect to the aforementioned scheme. The following schemes, which are similar to the static compilation one, are provided by ILDJIT: static, ahead-of-time, partial static and partial-ahead-of-time compilations. Those schemes are introduced in the next sections.

2.2.1 Static Compilation in ILDJIT

ILDJIT implements the static compilation scheme by executing two phases in one run: the compilation and the execution phase. These phases are kept separated. This type of execution is shown in Fig. 2.3, where ILDJIT has been invoked with the parameter `--static`. In this compilation scheme, ILDJIT does not interchange compilation and execution of the program: first, the entire program is translated to the target machine code (e.g. Intel x86). When the machine code has been generated, and also stored in memory, ILDJIT shutdowns the compiler both to free as much memory as possible and to reduce the overall number of running threads (the compiler is multi-threaded). When only the runtime module resides in memory, the execution of the produced code can start.

In order to compile our hello world program by using the static compilation scheme, we need to execute the following command:

```
$ iljit --static ./hello_world_c.cil
Hello, world!
```

In this example, ILDJIT both produces and executes the machine code of the underlying platform of the program `hello_world_c.cil`, which prints to the terminal the string `Hello, world!`.

In order to reduce the first phase, where the machine code is generated, ILDJIT stores the optimized code in its code cache, which resides in the file system at `~/.ildjit/manfred`, where `~` refers to the home of the user (e.g. `/home/simone`). Every program has its own directory inside the code cache called with the name of its correspondent CIL file. For example, in our previous example, where we compiled and executed our hello world program, the directory `~/.ildjit/manfred/hello_world_c.cil` has been generated. The parameter of ILDJIT `--clean-code-cache` removes every code inside this code cache.

```
$ ls ~/.ildjit/manfred
hello_world_c.cil
$ iljit --clean-code-cache
$ ls ~/.ildjit/manfred
$
```

The code generated and stored inside the code cache is platform independent. ILDJIT stores its intermediate representation inside this cache instead of the final machine code. The reason is that most of the time spent by the compiler is due to the code optimizations; these optimizations are mainly performed to the intermediate representation. By storing the already optimized intermediate representation to the code cache, at the second time a program is invoked, ILDJIT loads the code from its cache (generated at the first run), it generates the machine code and it starts the execution. This time, the time spent by the compiler is negligible in usual scenarios. For example, on an Intel core i7 machine, the overall time spent by the compiler at the second invocation to generate the machine code for an entire real program is few ms.

2.2.2 Ahead-of-Time Compilation in ILDJIT

Other than the already described static compilation scheme, ILDJIT provides the ahead-of-time (AOT) compilation. The main difference between the AOT and the static compilation scheme is that in the first one, the compiler is kept in memory even

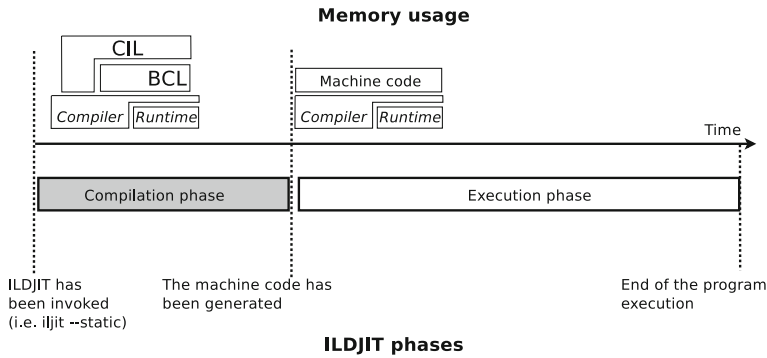


Fig. 2.4 Ahead-of-time compilation scheme implemented inside ILDJIT

after the compilation phase shown in Fig. 2.3. The resulting execution is shown in Fig. 2.4.

The AOT compilation has been introduced to compile an intermediate language, such as Java bytecode or CIL, into the target machine code before actually running it. However, recompilation of the produced code can happen at runtime in this scheme. These recompilations are typical of dynamic compilers, which exploit runtime information to produce better code. For this reason, ILDJIT keeps the compiler in memory even after the compilation phase.

2.2.3 Partial Compilations in ILDJIT

Sections 2.1 and 2.2 describes two static compilation schemes available in ILDJIT. In both cases, both the entire program and everything it is linked with are translated to the ILDJIT intermediate representation (IR) first and to the machine code later. Consider for example our hello world program, `hello_world_c.cil`. In this case, every method defined either inside the `hello_world_c.cil` file or inside the entire BCL is compiled.

On top of these two schemes, ILDJIT provides a partial compilation option, `-P1`, which can be used for both ones. When this option is specified, ILDJIT applies the chosen compilation scheme only to those methods effectively executed by a previous run of the program. This type of partial compilation is structured in three phases: first, we need to clear the code cache.

```
$ iljit -clean-code-cache
```

Second, the program is executed to keep track of which methods are needed.

```
$ ls ~/.ildjit/manfred
$ iljit -P1 ./hello_world_c.cil
Hello, world!
```



```
$ ls ~/. ildjit/manfred
hello_world_c.cil
```

Finally, the chosen compilation scheme is applied constraining it to the methods previously executed.

```
$ iljit -P1--static ./hello_world_c.cil
Hello, world!
$ ls ~/. ildjit/manfred
hello_world_c.cil
```

Later executions of the program do not need the `-P1` option anymore because the code has been generated already. Hence

```
$ iljit --static ./hello_world_c.cil
Hello, world!
or
$ iljit-aot ./hello_world_c.cil
Hello, world!
```

Notice that even if the second phase does not produce any code, after its execution, a code cache entry for the considered program has been created. Information inside the code cache at that stage includes profiling data only. Hence, no code is included. By looking inside the cache, we can see a file called `profile.ir`.

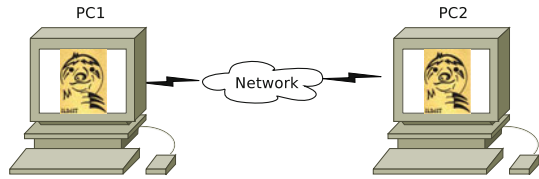
```
$ iljit --clean-code-cache
$ iljit -P1 ./hello_world_c.cil
Hello, world!
$ ls ~/. ildjit/manfred
hello_world_c.cil
$ ls ~/. ildjit/manfred/hello_world_c.cil/*.ir
profile.ir
```

On the other hand, after the third phase, the same code cache entry of the considered program includes code effectively. Hence, the third phase replaces the profile data with the generated code.

```
$ ls ~/. ildjit/manfred/hello_world_c.cil/*.ir
profile.ir
$ iljit -P1 --static ./hello_world_c.cil
Hello, world!
$ ls ~/. ildjit/manfred/hello_world_c.cil/*.ir
methods.ir
```

The file `methods.ir` contains the code of the methods specified inside the file `profile.ir`.

Fig. 2.5 IR produced inside one system and exploited by another one



2.2.4 Cached Code

As previously described, the code stored inside the code cache of ILDJIT (i.e., `~/ildjit/manfred`) contains the IR representation of the program. By default, ILDJIT stores platform independent code inside this cache only (this behavior can be changed by customizing the framework as described in [Chap. 6](#)). Since the cached code is platform independent, we can exploit it on different systems where either the underlying operating system or the platform can differ with respect to the one used to produce that code.

Consider for example the scenarios described by [Fig. 2.5](#) where two computers are involved: PC1 and PC2. The first one, PC1, has a Windows operating system installed on top of an Intel x86 processor. On the other hand, the second one, PC2, has Linux installed on top of an ARM processor. The code is first generated on PC1.

```
PC1 $ iljit --static hello_world_c.cil
Hello, world!
```

Then, the code cache is transferred from PC1 to PC2.

```
PC1 $ rcp -r ~/ildjit/manfred/hello_world_c.cil
PC1: ~/ildjit/manfred/
```

Hence, ILDJIT installed on PC2 can exploit the code produced by PC1.

```
PC2 $ iljit --static hello_world_c.cil
Hello, world!
```

Thanks to this platform independent property of the produced IR code, CIL programs can be compiled once and used inside every system ILDJIT has been installed into, no matter which system produced that code.

2.3 Dynamic Compilation

Software portability suggests the generation of portable intermediate binary code, that remains independent from the specific hardware architecture and is executed by a software layer called *virtual machine (VM)* [9]. A virtual machine provides an

interface to an abstract computing machine that accepts the intermediate binary code as its native language; in this way, the virtualization of the instruction set architecture (ISA) is performed. The dynamic compilation approach was introduced to overcome the slowness of the first generation of virtual machines, where the execution of bytecode programs was entirely interpreted: they interpreted bytecode rather than first compiling it to machine code and then executing the so produced code. This approach, of course, did not offer the best possible performance, as the system spent more time executing the interpreter than the program it was supposed to be running.

ILDJIT provides two different dynamic compilation schemes: the just-in-time (JIT) [9] and the dynamic-look-ahead (DLA) [2] one. In both cases, the code cache is not used. The second one, the DLA compilation, is a natural evolution of the JIT compilation specifically designed for the multicore era. These schemes interchange the compilation and the execution of the program leading to an interleaving of the corresponding phases shown in Fig. 2.3.

The code produced by dynamic compilers can be different with respect to the one produced by static compilers described in Sect. 2.2. The reasons are the following: first the compilation performed at runtime should be as fast as possible in order to reduce its overhead at runtime. Hence, the dynamic compiler cannot spend too much time to optimize the code. Second, the compilation performed at runtime can exploit runtime information, such as values of method parameters, not available at static time.

2.3.1 *Just-in-Time Compilation*

Strictly defined, a JIT compiler translates bytecode into machine code, before its execution, in a lazy fashion: the JIT compiles a code path only when it knows that code path is about to be executed (hence the name, just-in-time compilation). This approach allows the program to start up more quickly, as a lengthy compilation phase is not needed before execution beginning. Figure 2.6 shows the execution of ILDJIT by using this scheme. This figure shows that the machine code is generated during the entire execution of the program by interleaving execution and compilation phases. Hence, the memory used to store the produced code grows as the execution of the program proceeds. Several approaches have been proposed in literature [9] to constrain this memory growing effect by discarding code produced in the past which is likely to be not useful in the near future. By default, ILDJIT does not constrain this growing effect. However, as described in Chap. 6, ILDJIT can be customized to change this behavior.

The JIT compiler is the default compilation scheme used by ILDJIT. Hence, we do not need additional options to use it.

```
$ iljit hello_world_c.cil
Hello, world!
```

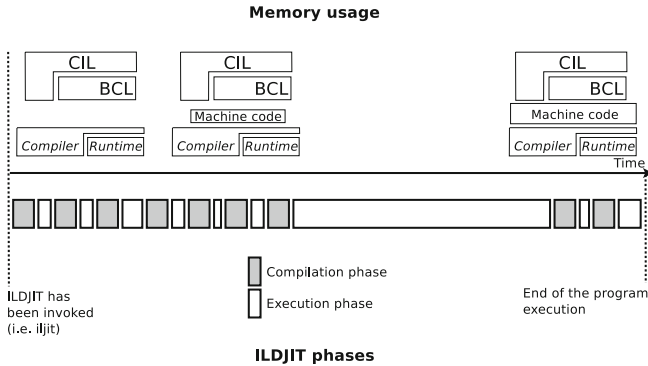


Fig. 2.6 Just-in-time compilation scheme implemented inside ILDJIT

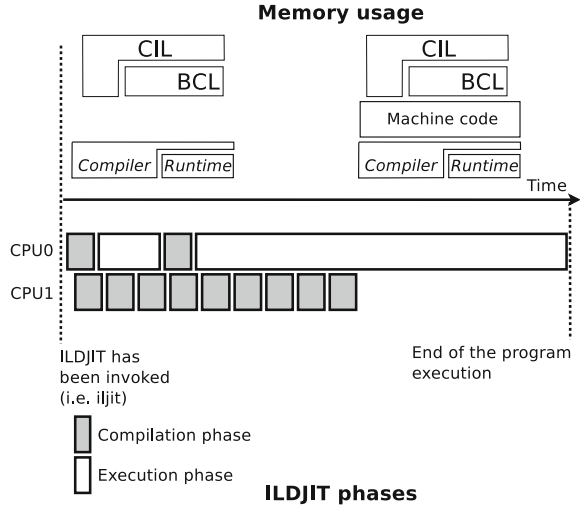
The JIT approach seems promising comparing it to the bytecode interpretation solution, but it presents some drawbacks: JIT compilation removes the overhead due to the interpretation at the expense of some additional startup cost, and the level of code optimization is mediocre. To avoid a significant startup penalty for portable applications, the JIT compiler has to be fast, which means that it cannot spend much time in optimization. The next section describes an evolution of this scheme, called Dynamic Look-Ahead compilation [10], which exploits the parallelism available in multicore architectures to both improve the quality of the code and to reduce the compilation time perceived by the execution of the program.

2.3.2 Dynamic Look-Ahead Compilation

Dynamically compiled code can achieve large speedups, especially in the long run, since the execution of a native method is an order of magnitude faster than that of an interpreted one. However, the performance of a JIT-based dynamic compiler is still lower than that of native code produced by static compilation schemes like the AOT one. The loss of performance is due to both compilation overhead, often called *startup time*, and to the poor quality of the generated code, since the startup time minimization prevents the aggressive and costly optimizations usually performed by static compilers.

Nowadays, multi-core technology has become the predominant technology in both desktop and embedded domains. This type of hardware architecture is a way to provide more computational power without relying on the reduction of the clock cycle, which is becoming increasingly difficult due to technology limitations. For this reason, dynamic look-ahead (DLA) compilers exploit multiprocessor environments by introducing compiler threads, which can dynamically compile bytecode portions in advance, in parallel with the application execution. Strictly defined, a DLA compiler translates and optimizes bytecode looking ahead of the execution

Fig.2.7 Dynamic look-ahead compilation scheme implemented inside ILDJIT



in order to both anticipate the compilation before the execution asks for it, and to produce optimized code.

DLA compilers are based on a software pipeline architecture for compilation, optimization and execution tasks. While a processor is executing a method, compilation threads (running on other processors) *look ahead* into the call graph, detecting methods that have good chances to be executed in the near future. Moreover, they guess whether a method is a *hot spot* [9] (code often executed) or not, and apply aggressive optimizations accordingly. In the best case, there is no compilation overhead, because compilation fully overlaps with execution and methods are already compiled when they are invoked. Moreover, optimizations are fully exploited to provide high quality code.

Figure 2.7 shows a typical execution of a DLA compiler. The main difference with respect to JIT compilers is that DLA compilers can translate more code than the JIT ones (because it compiles in advance methods by guessing where the execution is going). Moreover, the produced code is optimized already before its first execution.

In order to use the DLA compiler available in ILDJIT, the option `-dla` is provided.

```
$ iljit --dla hello_world_c.cil
Hello, world!
```

By default, ILDJIT exploits every core provided by the underlying platform whenever the DLA compiler is used and there is a peak in term of methods to compile to run the application.

2.4 Different Configurations with a Single Installation

ILDJIT is a framework that includes a set of modules, which compose its core, and a set of external plugins, which provide different type of translations, code optimizations, memory managements and policies used by various compilers previously described.

Different users can have different needs, which means that different sets of plugins have to be used by different users. ILDJIT comes with a default set of plugins, which are installed in the system, and therefore, they are shared between users of that installation. However, ILDJIT provides a solution, following described, of having personal customizations of a single installation of the framework. This can be useful both to override some default plugins and to add new ones.

The aforementioned solution applied to ILDJIT about personal customizations of the framework is based on environment variables called `ILDJIT_X_PLUGINS`, where X is the name of the specific extension that it refers to. At boost time, ILDJIT loads these customizations in the following order: first, it loads the plugins in the same order specified by the list of directories declared by the environment variable `ILDJIT_X_PLUGINS`. Finally, it loads plugins from the directory where it has been installed. Consider for example the following list of directories:

```
$ echo $ ILDJIT_X_PLUGINS
/home/simone/first:/home/simone/second
```

Assuming that ILDJIT is installed inside the default directory, which is `/usr/local`, then plugins are loaded and used in the following order: `/home/simone/first`, `/home/simone/second`, `/usr/local/lib/iljit/optim-izers`. Assuming that we have a task, like dead code elimination, which is provided by two different plugins: one provided by the default installation and one installed in `/home/simone/first`. In this case, by having the directories specified as for the previous variable `ILDJIT_X_PLUGINS`, ILDJIT will use the plugin installed in `/home/simone/first`. On the other hand, if we change the value of the variable `ILDJIT_X_PLUGINS` as following:

```
$ export ILDJIT_X_PLUGINS=/home/simone/second
```

then the plugin provided by the default installation will be used. In order to provide this behavior, ILDJIT links at runtime plugins found in the system, which provide tasks not provided by other plugins that it is not already linked with.

Consider a scenario where there are three users: `alex`, `bob` and `tom`. Consider also that ILDJIT has been installed inside the default directory and that these three users share the same system (or they share the file system at least). Imagine that `alex` and `bob` need to customize ILDJIT with their own plugins. Moreover, `tom`

needs both to customize ILDJIT with his own plugins and to use bob's plugins. In this case, alex needs to set the aforementioned variable as following:

```
$ export ILDJIT_X_PLUGINS=/home/alex/my_plugins
```

On the other hand, bob needs to set it as following:

```
$ export ILDJIT_X_PLUGINS=/home/bob/my_plugins
```

Finally, tom needs to set the variable to point to the following two directories:

```
$ export ILDJIT_X_PLUGINS=/home/tom/my_plugins:  
/home/bob/my_plugins
```

Notice that tom needs to have read access to the bob's directory in order to use his plugins.

References

1. Campanoni, S., Agosta, G., Crespi-Reghizzi S., Biagio A.D.: A highly flexible, parallel virtual machine: design and experience of ILDJIT. In: Software: Practice and Experience, pp. 177–207 Wiley (2010)
2. ECMA ECMA-335: common language infrastructure (CLI). <http://www.ecma-international.org/publications/files/ECMA-ST/ECma-335.pdf> (2010). Cited 11 June 2011
3. ISO (1999). ISO C Standard 1999
4. ISO (2003). ISO/IEC 14882:2003
5. Gosling, J., Bill, J., Steele, G., Bracha, G.: The Java Language Specification. 3rd edn. Addison-Wesley (2005)
6. ECMA ECMA-334: C# Language Specification. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf> (2006). Cited 11 June 2011
7. Costa, R., Ornstein A.C., Rohou, E. GCC4CLI. <http://gcc.gnu.org/projects/cli.html> (2010). Cited 11 June 2011
8. de Icaza, M., Molaro, P., Mono, D.M. <http://www.mono-project.com> (2011). Cited 11 June 2011
9. Smith, J., Nair R.: Virtual Machines: versatile platforms for systems and processes. In: The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers (2005)
10. Campanoni, S., Sykora M., Agosta, G., Crespi-Reghizzi S.: Dynamic look ahead compilation: a technique to hide JIT compilation latencies in multicore environment. International conference on compiler construction, pp. 220–235 (2009)

Guide to ILDJIT

Campanoni, S.

2011, XIII, 97 p. 30 illus., Softcover

ISBN: 978-1-4471-2193-0