SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **00**:1–36   *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

A highly flexible, parallel virtual machine: design and experience of ILDJIT[*]

Simone Campanoni[‡], Andrea Di Biagio[‡], Giovanni Agosta, Stefano Crespi Reghizzi

Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

SUMMARY

ILDJIT, a new-generation dynamic compiler and virtual machine designed to support parallel compilation, is here introduced. Our dynamic compiler targets the increasingly popular ECMA-335 specification. The goal of this project is twofold: on one hand, it aims at exploiting the parallelism exposed by multi-core architectures to hide dynamic compilation latencies by pipelining compilation and execution tasks; on the other hand, it provides a flexible, modular and adaptive framework for dynamic code optimization. The ILDJIT organization and the compiler design choices are presented and discussed highlighting how adaptability and extensibility can be achieved. Thanks to the compilation latency masking effect of the pipeline organization, our dynamic compiler is able to mask most of the compilation delay, when the underlying hardware exposes sufficient parallelism. Even when running on a single core, the ILDJIT adaptive optimization framework manages to speed up the computation with respect to other open source implementations of ECMA-335.

KEY WORDS: dynamic adaptation, CIL dynamic compiler, parallel virtual machine

1. Introduction

The use of a virtual machine language, instead of machine code, is by now a well-established and successful technique for porting programs across different hardware platforms, without incurring into the difficulties and draw-backs of software distribution, when done at source-language level. In addition, interoperability between different source languages is made possible by their translation into a common suitable virtual code. Java bytecode first,

[*]Supported in part by the European Commission under Framework Programme 7, OpenMedia Platform project
[†]E-mail: {campanoni,agosta,dibiagio,crespi}@elet.polimi.it
[‡]Supported in part by the ST Microelectronics. Corresponding author

Received 02/17/2009

and then *CLI (Common Language Infrastructure)* are the *de facto* industrial standards for such virtualization of the Instruction Set Architecture (ISA). In particular CLI, after its international standardization as ECMA 335 [22] and ISO/IEC 23271:2006, has become a very attractive framework, where applications written in multiple high-level languages can be executed in different system environments, at no cost for adaptation.

The increasing acceptance of CLI, also in areas traditionally reserved to direct compilation into machine code, is witnessed by the growth and consolidation over the years of the tool chains needed to support virtualization, namely static front-end compilers translating popular source languages into virtual code, and Virtual Execution Systems (VES), such as .NET, Mono, and Portable.Net. Although the first VES's were based on code interpretation, all modern ones use instead Dynamic Compilation (DC), in order to achieve better performances.

A VES is a very complex software system that takes years for his development and in a sense is never finished, since new requirements keep coming from the advances in machine architectures, source languages, operating systems and middleware. Therefore it is not surprising that the current generation of VES has been designed having in mind the desktop personal computers and the traditional embedded systems, before the advent of multi-core architectures and single-chip multiprocessors, and without taking into consideration the great opportunities they offer for parallelization inside the VES and between VES and application.

The new open-software VES that we present here is named ILDJIT (*Intermediate Language Distributed Just-In-Time* translator). In our opinion, it can be considered as a new generation of VES, especially intended for distributed and parallel architectures, and flexible enough to easily adapt to the evolving and hard to anticipate requirements of modern computer platforms. ILDJIT is designed to be easily extensible by providing a framework where existing modules can be substituted by user customized ones.

Our project from the very start in 2005 focused on multi-processors and aimed at offering a complete framework to study the different components of a dynamic compiler and their interactions in such environments. We believe our experience should be of interest to anyone considering porting or designing a virtual machine and dynamic compiler for a directly interpretable representation (such as CIL or Java bytecode) to a multi-processor architecture.

It is obvious that on a computer with more processors than application threads, the VES can run uninterrupted on a processor, so that many compilation/application balancing problems found on single processor machine simply disappear. According to Kulkarny et al. [31],

> *little is known about changes needed in balancing policy tuned on single-processor machines to optimize performances on multi-processor platforms. . . . it is a common perception that the controller could (and should) make more aggressive optimization decision to make use of the available free cycles. Aggressiveness in this context can imply compiling early, or compiling at higher optimization levels.*

We shared the same intuition, and, in order to take full advantage of the high degree of parallelism of future platforms, we designed the dynamic compiler as a parallel distributed program. Compiler parallelism is manifold. First, the compiler phases are organized as a pipeline so that several CIL methods can be simultaneously compiled by different compilation phases.

Second, since many kinds of optimizations are applied in static compilers, and it is not a priori known which of them are more effective in a dynamic setting, we decided to design separate optimizers as processes running on a common Intermediate Representation (IR).

Third, the VES software architecture ought to be flexible and open to an unpredictable number of new modules, as the project progresses and experience tells us which optimizations are productive for which applications. Moreover, flexibility is needed to choose the most performing solution from a set of alternative algorithms, depending on the application profile and needs: an example is garbage collection for dynamic memory allocation, where our system has four supports to automatically choose from. To obtain flexibility and modularity most modules are implemented as plug-ins. The entire system architecture is designed according to the software engineering principles of *design patterns* [24].

One could fear that a compiler implemented as a distributed and dynamically linkable program would have to pay a high overhead, to the point that the benefit from hardware parallelism might be offset especially for a small number of processors. On the contrary our experiments show that the performance of applications compiled by our system are comparable to some of the best known VES's, on a single processor machine, and superior on multi processor platforms. Of course we did benefit from part of the experience of earlier systems in such critical matters as granularity of compilation unit, unit compilation scheduling, optimization, memory management, and processor re-targeting.

We see our contribution to advancing the state-of-the-art of dynamic compilation as twofold. First we have addressed and solved several problems raised by parallel compilation, such as those caused by interdependencies between compiler threads and ensuing deadlocks, and we have designed an effective pipelined organization of compilation phases. Second we have carefully examined the existing know-how on VES techniques, and chosen and combined a set of well-matched methods. Whenever possible, we provide experimental data and qualitative reasoning in support of our design decisions.

The rest of this paper describes the salient features of the ILDJIT system, and is organized as follows. Section 2 summarizes the VES organization and introduces the main subsystems. Section 3 describes the multi-thread organization, the compilation pipeline, and focuses on interesting new problems raised by parallel compilations, in particular on a novel model (Dynamic Look-Ahead compilation [13]) for selecting the unit to be compiled. Section 4 describes the Intermediate Representation subsystem of ILDJIT. Section 5 introduces the ILDJIT adaptive optimization framework, and describes the optimization policies used in the current implementation. Section 6 provides the description of the module in charge to translate a CIL program into an equivalent IR program. Section 7 qualifies the different requirements for memory management, and compares several garbage collection algorithms used. Section 8 reports the current experimental comparisons of ILDJIT versus other comparable VES's. Section 9 briefly reviews the related works, including the dynamic compilers for CIL bytecode and other relevant tool-chains. At the end, Section 10 draws some conclusions and outlines the next steps of our research. It also mentions current applications of ILDJIT.

2. Overview of the ILDJIT Dynamic Compiler

ILDJIT implements the VES by means of a DC for performance reasons. The primary task of the VES is to translate each piece of CIL bytecode to a semantically equivalent target code to be directly executed by the hardware.

ILDJIT is designed with the key goals of flexibility, adaptability and modularity without compromising performance. In this section, we discuss the two primary design decisions that shape the software architecture of ILDJIT: the choice of the basic translation unit and the modular architecture of the system.

2.1. Translation unit

Choosing the correct granularity for the translation process is especially important in a dynamic compiler [19]. The function or method is generally taken as the natural unit [38]. A larger translation unit may provide additional optimization opportunities, but also imposes a higher risk of compiling code that will not be executed.

On the other hand, a smaller translation unit allows the compiler to output the translated code earlier, but heavily limits optimization, forcing the compiler to generate additional code to cope with frequent interruptions of execution. Specifically, if the unit is smaller than an entire function or method, then the computation state must be explicitly saved when switching between parts of a function that belong to different compilation units. In the other case this is not needed, since the function call boundary naturally defines a state to be preserved across a call (parameters and return values), while most of the local state can be destroyed (local variables of the callee).

In the case of CLI, another, language-specific, reason candidates the CIL method for the role of translation unit. The metadata stored inside each CIL bytecode file assume the method as the main compilation unit, so that most information (local variables, stack size) is stored in a method-wise fashion.

ILDJIT uses a flexible translation unit. The method is taken as the *minimum* translation unit. The choice of larger units is possible at runtime, on a case by case basis: a policy can be set in a dedicated policy module to decide the actual translation unit based on information obtained from metadata at object creation. Currently, it is used to detect components in component-based software and, if a component must be deployed to a different processor, consider it as a single translation unit.

In the rest of the paper, we will focus on individual methods, as they are the most common translation units.

Given this choice, there is a need to manage the invocation of not yet translated methods, when using lazy compilation [29], that is a method is only translated when one of its call points is first reached by the control flow. ILDJIT uses the traditional *trampoline* technique [29] to redirect such invocations to its own translation modules. After producing and installing the machine code translation of the invoked method, the call to the trampoline code is replaced with the actual memory address of the newly translated method.

Note that a trampoline has to be transparent to both caller and callee. While the caller is not supposed to perform any special check in passing parameters, the callee should not worry about the return value and the return address. The need for having different trampolines for different methods (as opposed to a single dispatch function in static compilation) comes from this principle.

2.2. Software Architecture

Here we provide an overview of the software architecture. ILDJIT is composed by the modules, or components, shown in Figure 1.

These modules can be divided into two groups: (1) the main components of the compiler system, including the Pipeliner, CLI Manager, Optimizer, Garbage Collector and IR Virtual Machine; and (2) the DC support infrastructure, including the profiling infrastructure (*Profiler*), the initialization subsystem (*Bootstrapper*), the data structure and support libraries (*Tools*) and finally the various policies (*Policies*). The latter ones will not be described in detail, being typical implementations of their class, with the exception of specific critical policies that affect the behavior of one of the main modules.

We provide a synthetic description of those modules whose name is not self-explanatory.

Pipeliner implements and manages the software pipeline needed for the translations, optimizations and executions of the CIL bytecode.

CLI Manager provides the functionalities needed to implement the CLI architecture and to translate the CIL bytecode to our IR language

IR virtual machine Translation of the IR code to the machine code and invocation of the latter.

Profiler Profiling functionalities for the internal modules of ILDJIT, as well as for the dynamically generated code .

Each module will be covered in its own Section, to highlight the specificities of ILDJIT with respect to typical dynamic compilers.

Finally, each module is further modularized by using the following approaches, depending on its typical use:

Dynamically loaded shared library This choice gives the highest degree of flexibility, allowing the module to be loaded at runtime. It is therefore employed for all components that can be freely replaced, added or removed from the system. This choice allows the implementation of a plugin framework, making the DC customizable for a specific application domain [37] (e.g. for multimedia components).

Statically loaded shared library Some components need to be present at all times in the system, yet, they are used by several subsystems that may run on different processors. This choice allows a good degree of flexibility, while removing unnecessary loading overheads [37].

Internal module Components that are not shared between subsystems are implemented as static libraries for maximum efficiency.

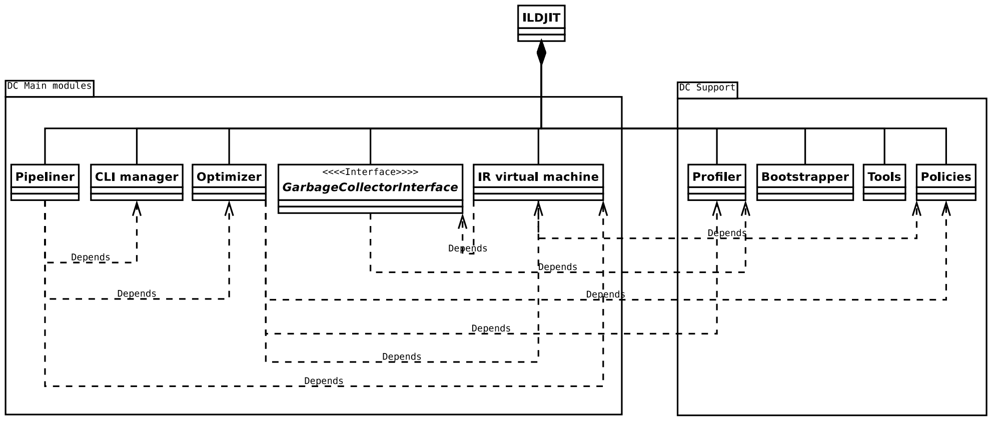


Figure 1. UML class diagram of ILDJIT software architecture. Dependences to the Tools, Profiler and Garbage collector are not shown (all modules depend).

3. Parallel Compilation

In traditional Just-In-Time compilers, the lazy compilation policy is adopted: when the control flow leads to a method not yet compiled, the application is suspended, and the method is translated. Given sufficient hardware resources, our DC can translate methods *before* the application invokes them, by parallelizing the application execution and dynamic compilation tasks. In this case, application execution does not yield to compilation. In the optimal case, the execution profile matches that of a statically compiled program.

Translation, optimization and execution of CIL code are managed by an internal software pipeline, designed not only to perform compilation and execution in parallel, but also to exploit pipeline parallelism between different compilation and optimization phases.

The pipeline model exposes five stages as shown in Figure 2 and explained in detail in Section 3.5. All stages can be parallelized, if hardware resources are available. Each stage can be performed by several parallel threads, possibly running on different CPUs, in order to simultaneously translate several CIL code pieces to IR. Similarly, several translation steps from IR to machine code may run in parallel.

The Pipeliner module (described in Section 3.5) adaptively controls the amount of computing resources allocated to the various dynamic compilation and optimization stages thus allowing ILDJIT to quickly react to variations in the workload. In case of light machine load, ILDJIT effectively hides the compilation overhead by ahead-of-time compilation of methods that are expected to be invoked soon. When the load is heavier, it can reduce resource usage by falling back to lazy compilation.

3.1. Dynamic Lookahead Compilation

A critical decision is which method to compile ahead of time. To tackle this issue, we have developed a technique termed Dynamic Lookahead Compilation [13]. It employs a combination

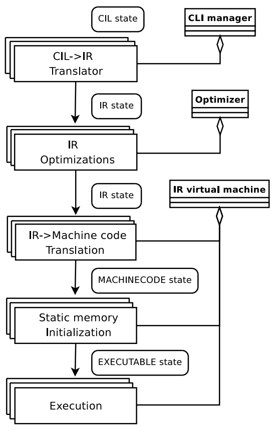


Figure 2. The translation pipeline model: on the left, the translation pipeline stages; on the right, the modules that implement the various stages; in the rounded boxes the state of the method code at the beginning of each stage

of criteria that estimate the likelihood that a method m_i will be invoked during the execution of method m_j, and estimate the distance between m_i and m_j as the shortest weighted path over the call graph between the two methods.

ILDJIT monitors the executing CIL method and computes the distance to other methods composing the application. All the CIL methods within a threshold D distance from the current method are candidates for translation and optimization. The rational of the distance based selection criterion is rather obvious: a method at near to zero distance will be probably required in the near future for execution, therefore it should be promptly translated to machine code to avoid a trampoline stall. Conversely, methods at greater distance do not need to be translated early, and the probability of them being requested in the future may be low. The threshold D is adjusted at runtime depending on available free CPUs'.

We define the *lookahead boundary* as the portion of the call graph of the application code, that includes the next candidate methods for compilation and optimization tasks. The lookahead boundary is a dynamic set, moving along with the executing method at a distance depending on the threshold D and available system resources (e.g., processing units dedicated to compilation tasks). An example of lookahead boundary and of its adaptation to the evolution of the execution of the application is shown in Figure 3.1; in this example the system resources available are considered constant as the threshold D.

As more CPUs become available, the lookahead boundary widens; a large boundary means ILDJIT compiles ahead of time many methods and then the probability of spending time inside the trampolines decreases – thus, the system appears to behave as if early compilation was employed instead of lazy compilation, but the compilation overhead is hidden by pipeline parallelism. An example of lookahead boundary and of its adaptation to the system resources available is shown in Figure 3.1.

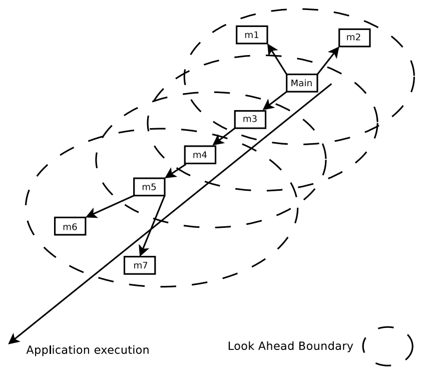


Figure 3. Adaptation of the lookahead boundary to the execution of the application code; the executing methods are in order: Main, m3, m4, m5; the threshold D is 2, constant.

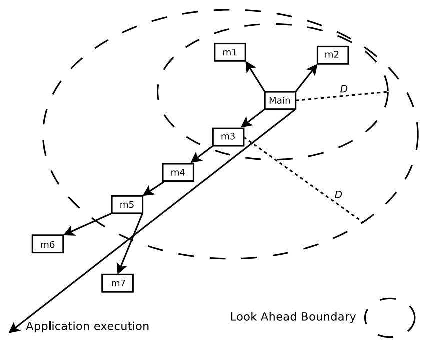


Figure 4. Lookahead boundary with adaptive threshold D; D evolves from 2 to 3; the executing methods are in order: Main, m3.

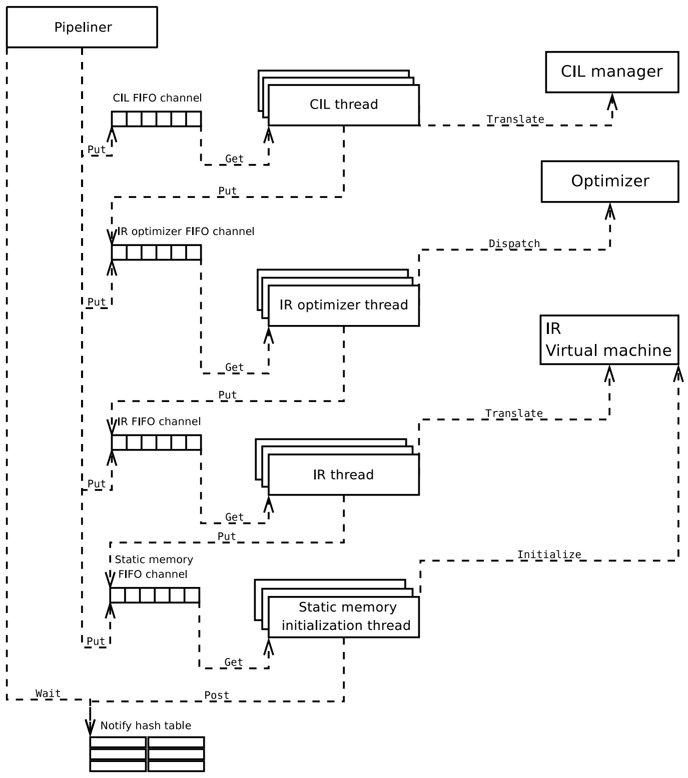


Figure 5. The Pipeliner structure

The Pipeliner module is organized in a set of threads depicted in Figure 5.

In the rest of this Section, we describes several issues that affect correctness and performance of a pipeline-based execution model and the implementation strategies adopted in the Pipeliner modules to tackle them.

3.2. Compilation load balancing

Load balancing is essential to obtain good performance improvements while minimizing the overhead. In the Pipeliner module, thread pools are used to minimize the overhead of the threading subsystem over the compilation time, and an adaptive strategy is employed to manage the size of the thread pools.

Four thread pools corresponding to the first four stages of the pipeline shown in Figure 2 contain respectively: the CIL to IR translation threads; the IR optimizer threads; the IR to

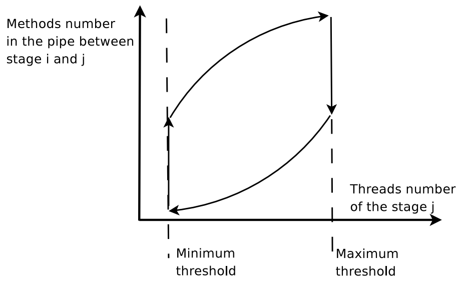

Methods number in the pipe between stage i and j
Threads number of the stage j
Minimum threshold
Maximum threshold

Figure 6. Pattern for threads adapting in the pipeline software

machine code translation threads; and the Static memory initialization threads. Each thread implements a single stage of the pipeline model working on a given translation unit.

The number of threads in each thread pool is adaptively chosen, taking into account the hardware resources and the current compilation workload. The number of threads ranges, for all pools, between a maximum and minimum threshold, chosen at deployment time to exploit knowledge of the underlying platform.

The Pipeliner dynamically adapts the number of threads assigned to each stage of the pipe, using the histeretic model shown in Figure 6. The number of threads of stage i depends only on the number of methods in the pipe between stages i and $i-1$. The choice of an histeretic model rather than a linear one is due to the non-negligible cost of creating and deleting threads, which makes a conservative policy more appealing.

3.3. Static memory initialization

In object-oriented programming languages, it is possible to define fields for both instance and class. The latter are static fields that are allocated at class loading time. Their content is implicitly initialized before the first access, and the values are preserved across method calls and class instantiation. In CIL, a class can define a special method `cctor` used to perform the initialization of the class fields.

In the Pipeliner module, the static memory initialization step is the final stage of the compilation pipeline. At this stage, the `cctor` methods containing for each static memory area the initialization operations are invoked directly from the Pipeliner.

When a method enters the compilation pipeline, a list of required `cctor` methods is fetched and appended to its description. When the method reaches the last stage of the pipeline, all `cctor` methods are fetched, and, if they have not yet been executed, they are compiled and invoked.

However, this mechanism is vulnerable to deadlock when it interacts with compilation load balancing based on a thread pool.

Table I. Method insertion policy into the compilation pipeline as function of the translation state

Translation State	Action
CILCODE	Append to the FIFO to CIL translation threads
IRCODE	Append to the FIFO to IR optimizer threads
MACHINECODE	Append to the FIFO to Static memory initialization threads
EXECUTABLE	Forwarded directly to the end of the pipeline

Consider the case of a set of classes $S = \{C_1, \ldots, C_{|S|}\}$, where $|S|$ is greater than the number n_t of threads available in the last stage of the Pipeliner. Assume that for all classes $C_i \in \{C_1, \ldots, C_{|S|-1}\}$, C_i contains the initialization of one of its own class fields using the value of a class field of C_{i+1}. Then, if C_1 is the first class to be referenced by a method m of a class C_0 not in S, thread T_0 (assuming the whole Pipeliner system to be empty at the beginning of the operation) will hold m, waiting for the translation and invocation of the `cctor` method of C_1, then thread T_1 will be used to held that `cctor` method. Consequently, each thread T_i, $i \in [0, n_t)$ will be used to held `cctor` method of C_i, and the whole system will exhaust the set of available threads, but the `cctor` method of C_{n_t} will need the compilation of the `cctor` method of C_{n_t+1}, since $n_t < |S|$. In this case, the whole system goes into a deadlock.

To avoid this deadlock issue, we do not use the load balancing mechanism described in Section 3.2 in the last stage of the pipeline.

3.4. Threads communication

The communication between consecutive stages of the pipeline is handled by asynchronous message passing, using software FIFO pipes allocated in shared memory between the threads of the Pipeliner module.

Even though items in each pipe are handled in a FIFO way, there is no guarantee that ordering of methods is preserved, since multiple threads are active at each stage, with potentially different execution times. However, out of order completion of the compilation process is not an issue except in specific cases, such as the static memory initialization, which are handled with ad hoc mechanisms, as shown in Section 3.3.

3.5. Pipeline entry

The Pipeliner takes as input a CIL method to translate. Depending on the current *method translation state*, the Pipeliner selects the correct entry point: e.g., if a method has been already translated to IR, then the translated method can be directly moved to the optimization stage. Table I reports the action taken in each method translation state.

The Pipeliner can be called synchronously or asynchronously. In the former case, it returns to the callee only when the input method is ready to be executed. In the latter case, the Pipeliner puts the method on the right FIFO, according to its translation state, and returns immediately to the callee.

Synchronous calls are performed by the Execution engine module of the IR Virtual Machine to translate methods that must be immediately executed.

Asynchronous calls are performed by the CIL → IR translator when a method call is found in the method under translation, and it is foreseen to be taken in the near future according to the distance estimation.

4. Intermediate Representation and Virtual Machine

Since most optimizations are done on the Intermediate Representation and the execution of the program is guaranteed by the *IR Virtual Machine* module, in this Section we provide details.

The current implementation of the IR Virtual Machine exploits a very high degree of modularity and task parallelism. Its design has been thought in order to improve adaptability and flexibility. As shown in Figure 7, the IR virtual machine is composed by six submodules which are further described in Section 4.1.

By applying the concept of separation of concerns, several steps of the translation process are delegated to submodules. In fact, the virtual machine implements the delegation pattern [24]: the single instance of IR virtual machine holds a reference to each submodule. During the translation process, the IR virtual machine orchestrates the interactions between its submodules by accessing to functionalities exposed via well defined interfaces. The life interval of each submodule may not exceed the life interval of the IR Virtual Machine.

As a result, of the design, the IR virtual machine can be implemented in a highly parallel and modular way. When feasible, submodules may work in parallel cooperating with each other. The computation of each submodule is triggered by explicit requests coming from the IR virtual machine instance.

Submodules are also responsible of: management of exceptional behaviors in the executed code; initialization of static memory; fetching information about IR data types and methods.

At runtime, the Pipeliner (see Section 3) module asks the virtual machine for the translation of IR methods interacting with the IR virtual machine interface. IR Virtual Machine internal modules are not visible from the outside.

4.1. IR Virtual Machine components

The translation from IR to the machine code is handled through the *IR→Machine code translator* interface, relying on the code generator installed within the system. Currently, ILDJIT provides the *Libjit Translator* module, which exploits Libjit [36], a code generator library.

In order to minimize the time spent for code generation, the IR language has been designed to be RISC like. The experimental evaluation given in Section 8 proves this design choice

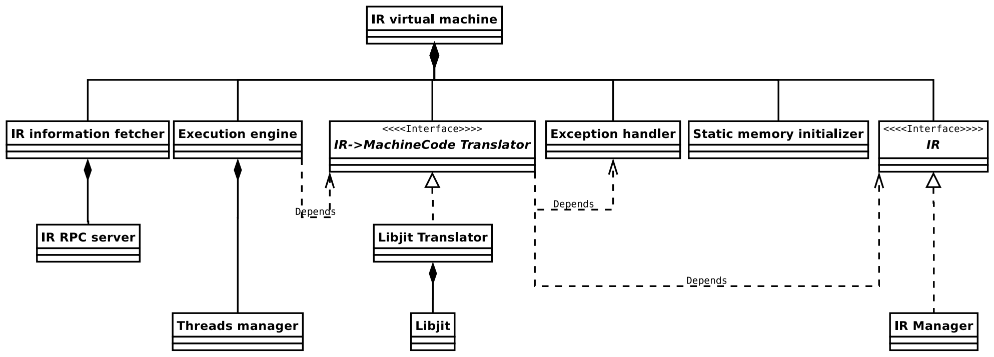


Figure 7. UML class diagram of the IR virtual machine

achieves its purpose. Moreover, *IR→Machine code translator* may enable machine dependent optimizations according to the policy provided by the optimization policy plugin 5.3.

The *Libjit Translator* module translates methods by simply interacting with Libjit through its API. Libjit provides currently supports x86, EM64T, ARM926 [40] and Alpha ISA.

The *Execution engine* module is in charge of executing the translated machine code. A method translated to machine code can be requested for execution by the IR Virtual Machine module. At runtime, the execution of a method can be paused to allow the translation of other methods through the use of trampolines as discussed in Section 2.1. A trampoline for a method that has never been compiled, implicitly triggers a translation request to the IR Virtual Machine. The main difference from a typical Just-In-Time compiler is that the translation and the execution phases are performed in parallel.

The Execution engine module includes the threading management capability which is provided by its *Threads manager* submodule. Currently the mapping between the IR and the OS threads used for them is bijective and it relies on the thread Posix library.

The *Exception handler* module supports the runtime exception handling mechanism. At runtime, a method may raise exception instances. Thus, the IR execution model gives the possibility to implement a unique handler routine for each method executed. When an exception is thrown at runtime, the execution is transferred to the handler of the function. The handler acts as a exceptions dispatcher: each exception instance is passed to a protected block of code either a `catch`, a `filter` or a `finally` block of code. If the function does not provide any protected region, the IR dispatcher ensures that the exception is propagated outside the method. The exception model is similar to the model specified in [32] albeit on our model there is only one exception handler for each method and there is not the possibility of bypassing the handler if an exception has been raised inside an exception block (what Lee called the *exc_signal* instruction). The implementation of the IR exception handling model is based on long jump machine code instructions.

The *IR information fetcher* retrieves information about IR data types and methods. The Optimizer module (see Section 5) interacts with the information fetcher module via RPC requests when information on data types and methods are needed.

The *Static Memory Initializer* calls each piece of code needed to initialize the IR static memory. Notice that ILDJIT ensures that the static memory is always initialized before its first use by using the software pipeline as its internal execution model (see Section 3).

Finally, the *IR Manager* module handles the modification of the IR methods (e.g. insert, remove, modify an instruction).

4.2. IR language

In this Section, we report the IR language.

The IR language is composed by 56 instructions classified in seven groups:

- Arithmetic instructions (e.g. `Add`, `Sub`, `AddOvf`)
- Memory instructions (e.g. `Load`, `Store`, `GetAddress`, `Length`, `Memcpy`, `InitMemory`)
- Object allocation instructions (e.g. `NewObject`, `NewArray`, `Alloca`)
- Call instructions (e.g. `Call`, `IndirectCall`)
- Compare instructions (e.g. `GT`, `LT`, `EQ`)
- Branch instructions (e.g. `Branch`, `BranchIf`)
- Exception instructions (e.g. `Throw`, `CallFinally`, `CallFilter`, `StartCatcher`, `StartFilter`)

The Arithmetic instructions are further divided depending on whether they can notify an overflow, by mean of an exception, or not. This distinction is important for the code optimization task because by this way, we know in advance, by the instruction type, if it can have side effects or not.

In the Memory instructions set, there is the Length instruction which is common in object oriented language and uncommon in intermediate representations. This instruction loads the length of the array into a register. From the semantic point of view, a Length instruction is a Load instruction with a fixed offset. We have specialized this semantic in order to implement aggressive code optimization algorithms like array bound checks removal.

The object and array allocations are specified, by the Object allocation instructions subset, giving the High level class information in order to be able to implement a precise memory alias analyzer.

The Exception instructions are further divided depending on whether they trigger an exception or they manage an occurred one. This distinction is important for a DC, because a reasonable policy can be implemented, where ILDJIT optimizes only IR instructions that do not handle an exception. In this way, the optimization time is less than a normal approach where the DC optimize the entire Control Flow Graph (CFG). The rationale of this choice is that exceptions are thrown infrequently and then the code that manage them will be executed infrequently.

5. Adaptive and Extensible Optimization Framework

Code optimization is a critical feature for a DC. On one hand, more optimization opportunities arise at runtime, on the other hand, optimization is often costly. Thus, balancing the costs and benefits of optimization is of capital importance in a DC.

ILDJIT optimizes both at IR and target code level. The rationale for this choice is that higher-level transformation are easier to apply when the semantic information of the source program is still available. For instance, array references are clearly recognizable, instead of being a sequence of low-level address calculations [9]. Machine-dependent optimization are, of course, performed at the target code level. An example of transformation that is useful at both levels is loop-invariant code motion, which can be applied both at IR level to expressions and at machine code level to address computations. The latter is particularly relevant when indexing multidimensional arrays [9].

In both cases, the transformation does not change the code level, so optimizing an IR method yields another IR method, and optimizing a target machine method yields another target machine method.

To allow the exploration of the cost-benefit trade-off, ILDJIT implements a modular and extensible framework where each optimization pass is structured as a plugin and it provides mechanisms for managing the dependencies between different optimization (and analysis) passes as well as interfaces for manipulating the intermediate representation (IR).

The use of plugins forces ILDJIT to rely on indirect call mechanisms to transfer information between optimization passes and intermediate representation. While this choice may seem suboptimal, we will provide experimental evidence in Section 8 to show that the performance degradation is not significant.

Finally, to maximize the benefits reaped from the availability of runtime information, ILDJIT implements adaptive optimization by means of an optimization policy plugin.

The rest of this Section describes the optimization framework as well as the optimization policy plugin.

5.1. Interfaces

To obtain good modularity, ILDJIT defines four interfaces, shown in Figure 8.

An internal module, *Optimizer*, is the bridge between the optimization plugin and the rest of ILDJIT.

The *IR Interface* (IRI) provides the optimization plugins with the ability to manipulate IR methods by adding, deleting, modifying or moving IR instructions.

The *Profiling Interface* (PI) allows to control the code profiling needed to supply information to the policy plugins.

The *Optimization Policy Interface* (OPI) defines the contract between ILDJIT and the optimization policy plugins.

Finally, the *Optimization Interface* (OI) is a goal-oriented, bidirectional interface that allows plugins to state their own requirements without declaring how to achieve them. This information is necessary to allow optimization plugins to rely on the results of analysis (or other

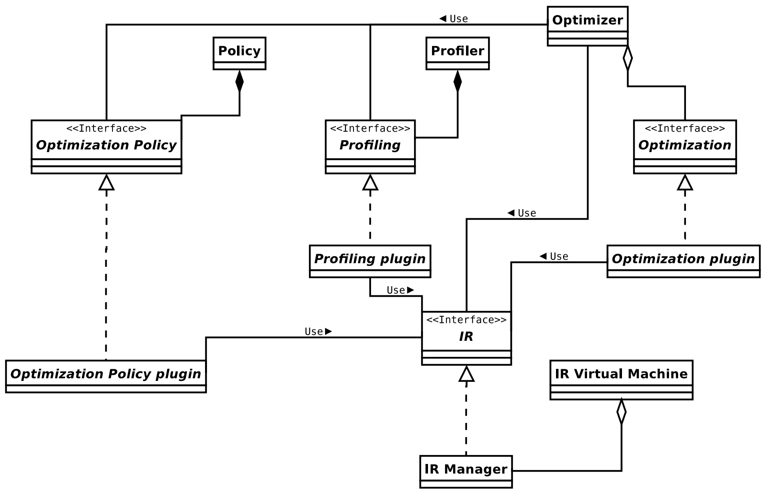


Figure 8. UML class diagram of the Optimization Interfaces and Modules

optimization) plugins. E.g., a forward propagation plugin may require data-flow information, without knowing which data-flow analysis plugin will compute it.

To this end, the Optimization Interface (OI) defines a set of high-level pre-defined optimization tasks (e.g. instruction scheduling) which are used by the external plugins to declare their targets as well as their dependencies. Through OI, ILDJIT can introspect optimization plugins, and, conversely, each plugin can request information from the DC. For instance, optimization plugins can request ILDJIT to invoke another plugin that satisfies a given dependency, such as computing available expressions, without knowing beforehand which plugin has the appropriate capabilities.

ILDJIT uses the information obtained through the OI to construct a dependency graph among the optimization plugins, and it verifies that all dependencies can actually be satisfied.

Figure 9 shows the dependency graph for the currently available optimization plugins.

5.2. Exploiting Runtime Information

With respect to a static compiler, a DC has the additional opportunity to exploit runtime information to improve some optimizations or enable new ones – e.g., runtime code specialization, inlining. To support these optimizations, the Optimization Interface exposes a second set of dependencies specific for runtime information, like variables values, parameters values, etc.

Currently, ILDJIT can provide runtime information about the dynamic call graph, runtime call stack, as well as statistics on the values of actual parameters and execution trace

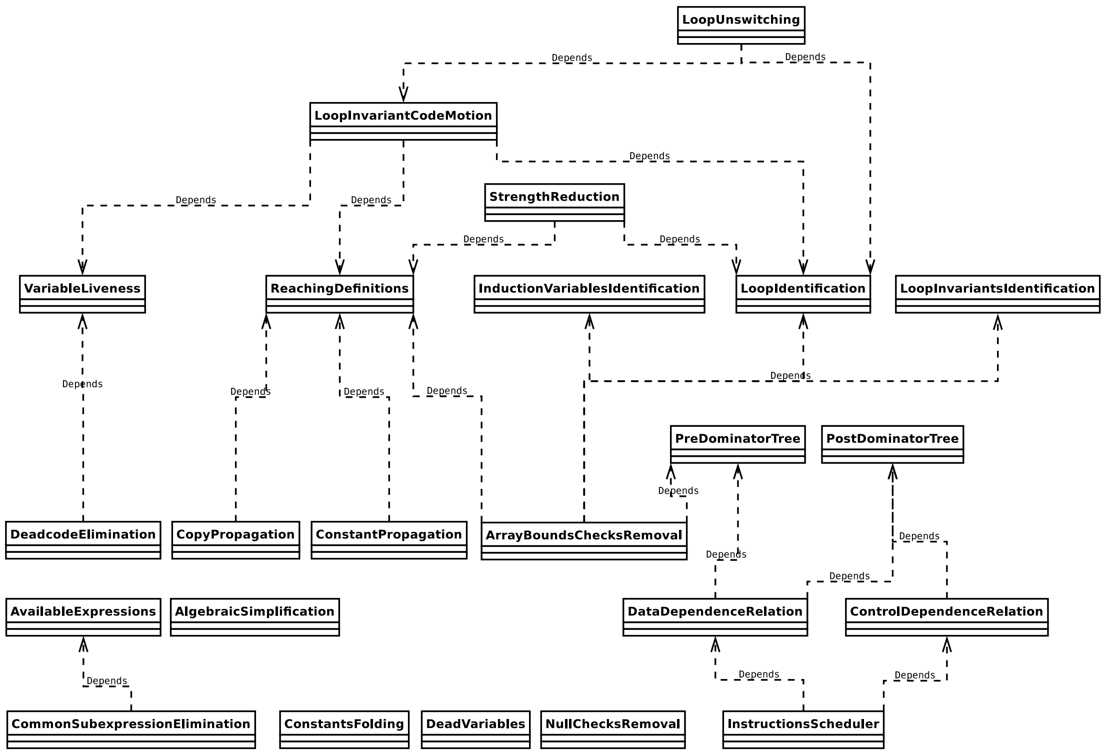


Figure 9. UML class diagram of the Optimization plugins and their dependences

frequencies. Plugins can choose the level of granularity of each item of information they request. For example, values of arguments may be flow-sensitive or flow-insensitive.

ILDJIT provides runtime information through the Profiling Interface (PI), which is used to interface with profiling plugins. This interface also provides introspection functionalities to discover the capabilities of each profiling plugin.

The key issue in balancing optimization costs and benefits is the choice of runtime information level of granularity. If it is too low, the cost of collecting and managing the information makes for a significant overhead, due to code instrumentation. If it is too high, the optimization benefits may be compromised by the lack of detailed information. In ILDJIT, optimization plugins control the level of granularity of the information they request, thus giving a high degree of flexibility, as well as locating the choice in the modules that include the domain knowledge needed to make the most cost-effective choice.

5.3. Optimization policy

A critical decision in dynamic code optimization is whether to apply or not an available transformation. Code transformations are by no means guaranteed to give a net benefit when applied to a given method, and some transformations may actually degrade performance when applied to the wrong code fragment. Even those transformations that cannot worsen the execution time may still pay an optimization overhead exceeding all the benefits.

A specialized plugin, the optimization policy manager plugin, is therefore put in charge of deciding which optimizations should be applied to each compiled method. It uses a set of policy plugins that implement specific heuristics, and interact through the OPI with the ILDJIT internal optimizer module, which mediates the communication. The same interface is also used to convey requests to ILDJIT for specific method information, which are then handled through the usual optimization and analysis plugins.

The current version includes a default policy plugin, which simply prioritizes optimizations in four levels:

1. Copy-propagation, algebraic simplification and dead-code elimination;
2. Constant-folding, constant propagation, dead-variables elimination and null check removal;
3. Loop-invariant code motion and strength reduction from multiplication to addition;
4. Trace rescheduling, array bounds checks removal, induction variable elimination and loop unswitching.

A heuristic policy, shown in Figure 10 is used to associate each method to an optimization level. Level 1 is associated to methods that are due for immediate execution, to avoid compilation delays as much as possible. Methods composed by a single loop are assigned to level 2, if the loop has only a small number of IR instructions, to level 3 otherwise. In all other cases, level 4 is used.

Multiple plugins can be used to implement an orchestration of heuristics, since a policy plugin can invoke other plugins.

5.4. Optimization channels

ILDJIT also provides remote optimization support across an IP network to allow slow clients to connect via Sun RPC [37] to remote optimization servers and obtain optimized code, in addition to more traditional local optimization. To preserve modularity and symmetry, local optimization is also handled by optimization servers, but the transport is provided by means of shared memory rather than RPC.

The choice of the most suitable Inter Process Communication (IPC) method is handled directly by the optimizer module. If sufficient local resources are available, then shared memory is used and a new optimizer server is launched. Otherwise ILDJIT attempts to use a remote machine chosen from a list defined at configuration time.

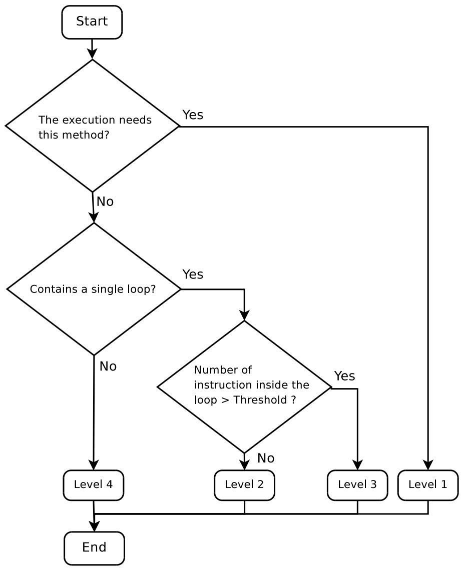

Start
The execution needs this method?
Yes
No
Contains a single loop?
Yes
No
Number of instruction inside the loop > Threshold ?
Yes
No
Level 4
Level 2
Level 3
Level 1
End

Figure 10. Optimization Policy. The methods are considered differently depending on: whether they contain or not a loop, whether they contain multiple loops, and finally on the number of instructions in a loop.

6. CLI Metadata Management

The CLI manager module (Figure 11) is in charge of managing the CIL bytecode according the ECMA-335 specification, to:

- Load and decode CIL files;
- Translate CIL methods in an intermediate representation (our IR language);
- Recognize, decode and create instances of built-in and user defined value types;
- Inspect metadata tables in order to retrieve memory layout informations for both CIL classes and value-types;
- Support the CLI exception handling model;
- Provide the implementations of various internal calls of the C# base class library.

Note that ECMA-335 identifies two subsets of the Common Language Infrastructure (CLI): the *managed* and the *unmanaged* code. Our experience is that implementing the support for both of them makes the project complexity much larger than the sum of the two isolated cases. In our experience, managed and unmanaged support require 2 and 1 person-years respectively, when developed independently, but their integration brings the total development time to 7 person-years. More precisely, the following issues should be taken into consideration:

- ECMA-335 allows objects to be moved from the managed to the unmanaged code – in this case, the GC must not consider that object anymore;

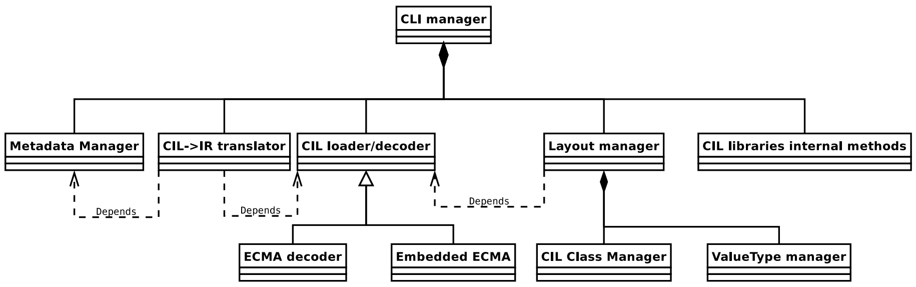


Figure 11. UML class diagram of the CLI manager software architecture

- To support aggressive optimization as well as precise garbage collection, the DC must take into account whether a pointer points to an object, a field, or an unmanaged data item, which increases the complexity of the internal type set.
- The DC has to provide facilities to support transition of the execution between the two subsets, including marshaling and unmarshaling of the parameters and platform dependent invocation.

6.1. Load and Decode tasks

Loading and decoding of CIL files is performed by external modules (*plugins*) that implement an interface. In addition to the modularity benefits inherent in the use of interfaces, plugins provide adaptability and extensibility to support, e.g., non-standard extensions to the CIL bytecode produced by specific front-end compilers.

Currently, ILDJIT includes two loading and decoding plugins, *ECMA* and *Embedded ECMA*. The two differ in the data caching policies: the *Embedded ECMA* decoding plugin implements a more restrictive policy, only caching information while its memory occupation remains under a given (target machine-dependent) threshold, thus trading off memory space for computation time, while the basic *ECMA* plugin employs a more aggressive caching policy, suited to high-end targets.

The two plugins share a large subset of functionalities, which are factored into the *Decoding Tools* library. This library supports the decoding of all the *metadata* defined in the ECMA-335 standard, intended to describe the internal structure and the content of a CIL file.

The *Type Checker* module supports the inspection of metadata information stored inside CLI files and referenced through *metadata tokens*. During the translation phase, the CLI manager frequently needs to retrieve metadata information for value and reference types, which represent semantic information behind the type indentifier (e.g., which interfaces a class type implements, or whether a type is an interface or a class).

Basically, *Type Checker* provides functions that can be used to check whether a property is held by a given type.

Type Checker has been implemented in a highly modular way, hiding the implementation logic behind the easy-to-use interfaces. The behavior of each function exposed by the interface can be redefined in order to allow adaptability and flexibility. For example, it is possible to implement metadata caching policies for systems where memory space is critical, and not all tables can be kept in memory at the same time.

Thus, multiple *Type Checker* module instances can be provided, with a runtime selection of the instance actually used. At the present time, this decision is performed implicitly when the *ECMA* plugin or the *Embedded ECMA* plugin are loaded into the system.

Moreover, Type Checker can be extended to support future CLI formats containing custom metadata tables, and is the only part of the compiler that needs to be modified to this end.

6.2. Layout Manager

ECMA-335 [22] standardizes the layout of data structures in a memory managed heap. To create valid instances of a given class or value type in memory, the DC needs information from the `ClassLayout` metadata table, accessed through the Type Checker.

The `ClassLayout` table holds an entry for each type that requires special constraints on how fields of valid instances should be laid out in memory. If this entry is not present for a type, then the loader module is free to decide how to layout in memory instances of that class.

To accomplish its main goal, the Layout Manager interacts directly with the loading and decoding plugin as well as the Type Checker module, and computes the relative offset of each field with respect to the header of the instance in memory. It also defines a *virtual table* containing all the references to method functions implemented by a class or value type.

Layout Manager behaves as an interface between the *IR Virtual Machine* and the memory layout information. The Layout Manager decouples functional issues (the correctness of the translation process performed by IR Virtual Machine) from performance issues, i.e., the trade-off between computing on-demand and caching layout information for rarely used classes and types. The on-demand computation of rarely used information is especially important for memory size optimization in the compilation of CLI programs, which can access very large libraries such as the `mscorlib` core library – caching layout information for all types defined in such libraries would lead to huge memory occupation. The Layout Manager therefore implements different caching policies tailored to the usage patterns of classes and types.

The interaction of Layout Manager and Garbage Collection is limited to the communication of the object size to the Layout Manager. Thus, the Garbage Collector interface and implementation are much simpler, since all layout issues are handled in a separate module.

To guarantee maximum performance, when there are no constraints on layout imposed by CLI to preserve low-level source language semantics, it is possible to dynamically redefine the layout policy. This requires that cached information for live objects be preserved across the redefinition, but allows to progressively change the policy as old objects are deallocated. Layout Manager thus implements the *strategy* design pattern [24].

When the IR Virtual Machine needs to create instances of value types on the stack, it interacts with a subcomponent of the Layout Manager, called *ValueType Manager*. The IR language implemented by the IR Virtual Machine knows how to allocate *Built-in Value Types* on the stack, since their size is known, but does not have the same kind of information about

User Defined Value Types. Thus, the IR Virtual Machine exposes an interface to explicitly manage the allocation of data elements onto the execution stack, and calls the ValueType Manager to perform this task.

6.3. Exception Manager

This module supports the creation of CLI exception instances and implements the exception handling mechanism. The current implementation is based on long jump mechanism.

The Exception Manager gets from the Type Checker the metadata for predefined exceptions (*CLI exceptions*) as well as the handler information, and packages them to speed up the generation of exception instances. Exception instances encapsulate information on the *exceptional behaviors* that affect the control flow, including a call stack image describing the site where the exceptional behavior was generated.

The module services are used by the IR Virtual Machine to raise exceptions and to trigger the execution of the exception handling procedure.

Finally, the exception handling mechanism directs the routing of the exception instances at runtime to the handlers. This operation is non-trivial, since ECMA-335 allows multiple handlers per method, while the IR Virtual Machine supports the definition of at most one exception handler per IR method. To bridge this gap, the IR exception handler must explicitly implement the control logic needed to dispatch an exception instance to the correct piece of IR code that implements the handler.

`System.OutOfMemoryException` instances are used to notify that there is not enough memory at runtime to fulfill a request of allocation for a new object into a managed memory heap. In this case, a `System.OutOfMemoryException` instance is created and implicitly thrown by the IR Virtual Machine.

In our opinion, even though the `System.OutOfMemoryException` is classified as an exception in the ECMA-335 specification, it is more a runtime error than a typical exception. Moreover, its peculiarities make it difficult to manage it properly. Indeed, the creation of an instance of this exception requires a memory allocation, which may result in another out of memory exception, leading to an endless loop. This issue is well known in Java virtual machine implementations as well. In some JVM implementations [5], garbage collectors fail to return a valid reference when the exception instance is created, leading to catastrophic errors.

A possible solution consists of handling a `System.OutOfMemoryException` as an error, avoiding the endless loop issue, but would not result in a compliant implementation of the ECMA-335 exception handling model, where all exceptions are catchable.

Currently, IR Virtual Machine provides the following behavior. A request of memory allocation for a new instance in memory is always forwarded to the garbage collector. If the memory allocation fails, the runtime system triggers the creation of a `System.OutOfMemoryException` instance. This assumes there is enough free memory to create a `System.OutOfMemoryException`. If this is not the case, the IR Virtual Machine does not raise a second exception instance, returning instead a singleton instance of `System.OutOfMemoryException` pre-allocated at bootstrap time.

As long as `System.OutOfMemoryException` can still be catched there is a strong possibility that other by memory exceptions be thrown out from the handler code. In this case, not

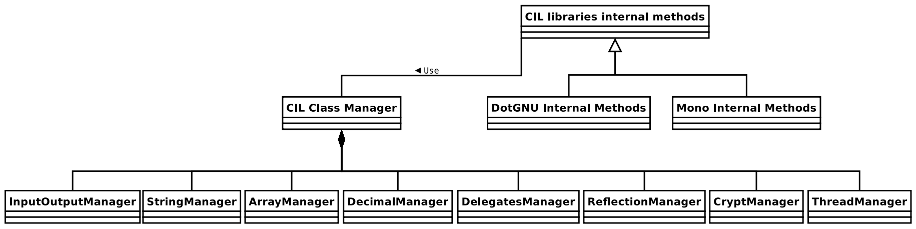


Figure 12. UML class diagram of the CLI Internal Methods module

providing a distinct `System.OutOfMemoryException` instance for each exception raised means there is no way to produce always a valid stack trace – which is required by the specification.

However, having pre-allocated an instance of `System.OutOfMemoryException` at bootstrap time, the IR Virtual Machine prevents the endless creation of out of memory exceptions, while introducing only a minor discrepancy from ECMA-335.

6.4. Internal Methods

In ECMA-335, a CIL method is said to be *internal* if only its signature is present in CIL language, while the body has to be supplied by the DC. A static compiler, when it generates the CIL assembly for a given source program (e.g. a C# program), must mark internal methods with the appropriate tag. Tags are needed by the DC for linking purposes.

Any static compiler must choose an implementation of the Base Class Library (BCL) (which is a set of CIL classes described in the ECMA standard) and therefore implicitly chooses the set of internal methods to be supplied by the DC.

As a consequence, the CIL programs are to some extent *Compiler-dependent*, or more precisely, *BCL-dependent*. Moreover, since the bodies of the internal methods have to be supplied by every DC, then they are replicated on them.

ILDJIT addresses this problem by providing the *CIL libraries Internal Methods* module, whose class diagram is shown in Figure 12. Our solution is based on the *CIL Class Manager* module, which supplies the functionalities needed to implement the DC-dependent actions, like allocation of a new object or determination of the offset of a given field of a CIL class. In this way, each BCL can be supported by ILDJIT providing a new sub-class of *CIL libraries Internal Methods*.

All sub-classes of *CIL libraries Internal Methods* are BCL-independent and provide the bodies of the internal methods. Whenever they need to do some DC-dependent action, they rely on the class *CIL Class Manager*.

Currently, ILDJIT provides two Internal Method classes: *Mono Internal Methods* and *DotGNU Internal Methods*. The names show which BCL they target.

7. Memory Management

The garbage collector (GC) module provides a range of memory management functionalities. The simplest service is to allocate memory portions to programs at their request, and to free them for reuse when no longer needed. ECMA-335 allows a memory portion to be declared no longer useful automatically or explicitly. Memory requests may be classified as: first, memory for the internal modules of the dynamic compiler; second, memory for the application program, which therefore contains only instances of CIL classes.

The two categories expose different characteristics. A chief difference is that, while the memory used for the DC can be explicitly freed by ILDJIT, the memory used for CIL objects can be automatically freed, without any explicit notification coming from execution of the application. Thus and other differences motivate our split of the memory manager into two tasks, each one applying different algorithms.

In order to experiment several different GC's, we rely on external modules.

An interface called *GarbageCollectorInterface* exposes the methods that each GC plugin has to supply; such plugins may be used for both memory sets, but two separate heaps are used for them. The interactions between the IR virtual machine and the GC are bidirectional. The garbage collector needs to access the IR virtual machine for the following reasons:

- to request the computation of the root set [42];
- to find the objects referenced by the current one;
- to check if an object can be moved from a memory location to another one (some objects may be constrained [22]);
- to call finalizer methods for the objects marked as garbage by the GC itself.

The above tasks are demanded to the VM because the GC does not known:

- the stack frame of the methods in order to compute the root set;
- the layout of the objects to compute the list of objects reachable at one step by a generic object;
- how-to translate and execute a CIL method.

These tasks can be performed by the IR virtual machine (see Section 4).

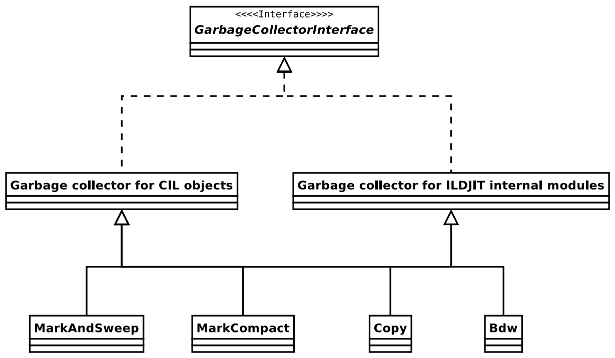
All memory allocations are recorded by the garbage collector to speed up the collection and make it precise [42]. Each garbage collection plugin manages the memory using a heap with a fixed size set by the bootstrapper module (see Section 2.2). The root set is managed using a dynamic array of pointers.

Since memory management greatly affects performances, and the effectiveness of different garbage collection algorithms varies depending on the memory access patterns, ILDJIT includes several garbage collection plugins. Currently, the user can choose among the following plugins, as shown in Figure 13:

MarkAndSweep implements the mark and sweep algorithm [42];

MarkCompact implements the mark compact GC algorithm [42];

Copy implements the copying GC algorithm [42];



<<<<Interface>>>>
GarbageCollectorInterface
Garbage collector for CIL objects
Garbage collector for ILDJIT internal modules
MarkAndSweep
MarkCompact
Copy
Bdw

Figure 13. UML class diagram of the Garbage collector module

Bdw is a wrapper for the GC of Boehm, Demers and Weiser [11], which is based on the mark and sweep algorithm, but supports extensions for generational GC.

The memory management system can be configured by the user to employ a different GC for application data than Bdw, which is always used for internal data. When used for internal data, Bdw is configured to perform non-generational collection, since the allocation and deallocation patterns of the ILDJIT internal data are simple enough that the more complex policy would not give any benefit. For the instances of CIL classes, on the other hand, the default configuration uses MarkCompact. MarkCompact has the advantage of improving locality with respect to MarkAndSweep or Bdw, without increasing the memory requirements as the Copy collector does.

8. Experimental evaluation

In this section we provide an experimental evaluation of both the pipelined compilation framework and the adaptive optimization framework implemented by ILDJIT. Moreover, we compare our dynamic compiler to its direct competitors, Mono [16] and Portable.Net [36], on two popular hardware platforms, x86 and ARM.

8.1. ILDJIT Implementation Details

ILDJIT is written in C using an object-oriented style. Although object-oriented languages such as Java, C++ and C# would provide more support for this programming style, and for our flexible building block architecture, we elected to use C for the following reasons: C coded by skilled programmers is more performant, it does not need a complex runtime and C compilers are readily available for most CPU families, an asset for portability.

The current version 0.1.0 comprises approximately 304 000 lines of C code, in about 1000 files organized in 45 external plugins, 6 shared libraries and 22 internal modules. The large number of plugins is due to the modular and extensible design of ILDJIT, and especially of the optimization framework.

The ILDJIT dynamic compiler is a long term software project, having been developed in about 40 months and with numerous ongoing developments [*]

8.2. Benchmarks

To evaluate ILDJIT and to compare it with its competitors, we used two different sets of benchmarks, one for managed CIL bytecode (obtained from C# benchmark programs) and the other for unmanaged code (obtained from C benchmark programs).

The first set include C# programs from the well known Java Grande [12] benchmarks suite as well as from the Computer Language Benchmarks (CLB) suite [1].

Table III summarises the C# programs. The number of methods reachable from the entry point of a program (Reachable CIL methods) is related to the size of the call graph. This value coupled with the information on the number of CIL methods effectively executed (CIL methods to translate) says how much of the reachable code is effectively needed to execute the program. The higher the ratio of the size of the call graph over the number of CIL methods to execute, the higher the probability of fruitlessly optimization (we remind that the time spent optimizing the code potentially adds to the total execution time of the DC).

Moreover, the smaller the average number of CIL opcodes per method, the higher is the importance of inter-procedure code analysis.

Finally, the number of CIL classes and the number of cctor methods is a good indicator on the effort spent by the dynamic compiler for the computation of memory alignments (also called classes layouting).

For unmanaged CIL programs, the benchmark suite CBench [23] is used. Collective Benchmark (CBench) is a collection/extension of open-source programs and multiple datasets to enable realistic benchmarking and research on program and architecture optimization. CBench includes the well known benchmark suite MiBench [26]. The characterization of the CBench programs is reported in the Table IV.

The CIL intermediate bytecode has been generated using the Portable.NET C# compiler (`cscc`) [36] for C# programs and `gcc4net` [15] for C programs. In all cases, `-O2` optimization was used.

The input of the benchmarks are taken from [12], [1] and [26]. For the JavaGrande benchmarks suite, there are three different input sizes: small (S), medium (M) and large (L). For the MiBench suite, there are two different input sizes, small (S) and large (L). Since the behavior of the virtual machines for M and L are close, we have chosen not to consider the input M. Finally, for the Computer Language Benchmarks there is only one input size.

[*]The experimental results reported herein represent a snapshot of the evolution of the systems considered. Results presented in this Section have been produced using *ILDJIT 0.1.0*, *Mono 2.4*, and *Portable.NET 0.8.1*.

Table II. Descriptions of programs written using the Object Oriented programming paradigm

Program	Description
	Java Grande benchmark suite written in C# language
FFT	Performs a one-dimensional forward transform of N complex numbers.
RayTracer	This program measures the performance of a 3D ray tracer. The scene rendered contains 64 spheres, and is rendered at a resolution of N×N pixels.
Montecarlo	A financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset.
SOR	Performs 100 iterations of successive over-relaxation on an N×N grid
Linpack	Solves an N×N linear system using LU factorization followed by a triangular solve.
SparseMM	Performs matrix-vector multiplication using an unstructured sparse N×N matrix stored in compressed-row format with a prescribed sparsity structure.
MolDyn	A simple N-body code modelling the behavior of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.
Euler	Solves the time-dependent Euler equations for flow in a channel with a “bump” on one of the walls.
	Computer Language Benchmark written in C# language
Fannkuch	Indexed-access to tiny integer-sequence N (see [6])
Mandelbrot	Generate Mandelbrot set portable bitmap file N (see [18])
Fasta	Generate and write N random DNA sequences
N-Body	Double-precision N-body simulation
SpectralNorm	Eigenvalue of N using the power method (see [20])

Notice that some benchmark suites evaluate the execution of portable multimedia applications on multi-processor embedded systems.

8.3. Impact of code optimization

ILDJIT is able to optimize the generated code before and/or after its execution. This Section provides a comparison of the execution time of ILDJIT when it applies code optimization algorithms described in Section 5.

In this Section we consider scenarios where the dynamic compiler chooses statically or dynamically the optimization level to apply during the code generation pass (see Section 5). In each scenario, the code optimizations are applied before its first execution; hence they do not exploit profile information.

ILDJIT1, ILDJIT2, ILDJIT3 and ILDJIT4 refer to the execution of ILDJIT when every method is compiled using the optimization level 1, 2, 3 and 4 respectively. Moreover, ILDJIT0 refers to the execution of ILDJIT when no code optimization algorithm is applied to the generated code. On the other hand, ILDJIT-Auto is the scenario where the optimization level is chosen at runtime leveraging on the policy described in Section 5.

Table III. CIL methods and classes composing the programs

Program	Reachable CIL methods	Reachable Internal methods	CIL classes	CIL methods to translate	cctor methods executed	CIL opcodes per method
FFT	1368	124	77	72	5	23
RayTracer	1609	131	90	138	5	27
Montecarlo	1503	129	82	96	5	28
SOR	1526	129	82	96	4	29
Linpack	1534	130	79	109	5	33
SparseMM	1535	129	81	105	6	25
MolDyn	1538	132	82	109	5	40
Euler						
Fannkuch	1493	129	77	90	4	29
Mandelbrot	1492	129	77	89	4	28
Fasta	1520	131	79	103	5	29
N-Body	1502	130	80	97	4	30
SpectralNorm	1521	130	79	94	4	28

Table IV. CIL methods and classes that compose the programs including the CIL code included by external libraries

Program	Reachable CIL methods	Reachable Internal methods	CIL classes	CIL methods to translate	cctor methods executed	CIL opcodes per method
Susan	1431	148	172	145	7	76
Jpeg	1589	148	271	265	7	79
Dijkstra	1415	146	158	138	7	53
Blowfish	1400	147	159	130	7	63
Rijndael	1417	147	170	142	7	88
SHA	1413	145	158	138	7	55
Adpcm	1392	144	157	124	6	46

Figure 14 shows the comparison among these scenarios. This Figure shows that the benchmarks with input size (S) run faster if no time is spent by the dynamic compiler to optimize the code. This is because the generated code is not executed enough times. On the other hand, for every benchmark with input size (L), the total execution time decreases as the code optimization level increases. Finally, we can observe that ILDJIT-Auto performance is close to ILDJIT4. The rational on this result is that our dynamic compiler has been tuned for the multi-core architecture; hence ILDJIT can distribute the optimization cost among CPUs and therefore it can be more aggressive than a normal dynamic compiler. On the other

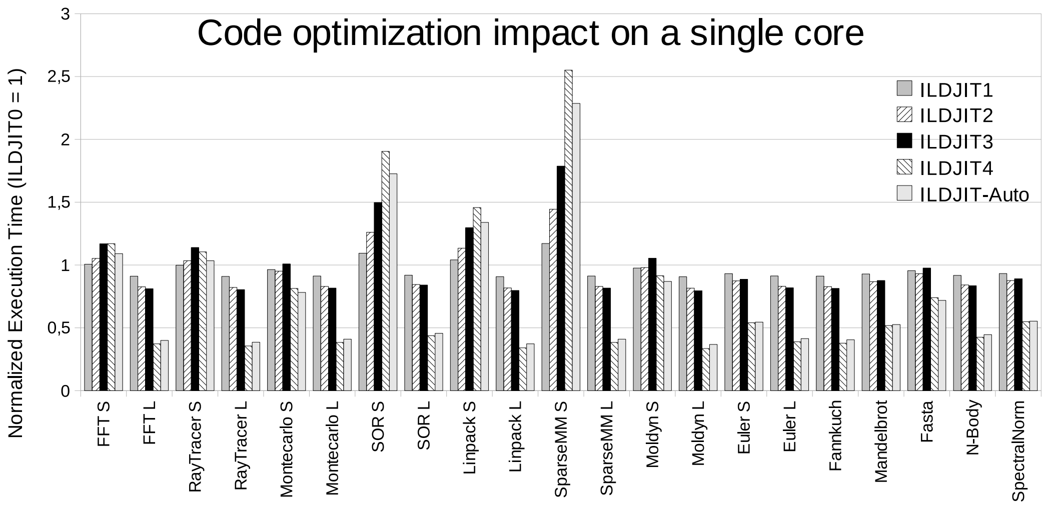


Figure 14. Total execution time spent by the various policies that choose the optimization level to apply before the execution of the generated code. Results are normalized to the execution time spent by ILDJIT without applying any optimization algorithm. Results are computed using one core

hand, when it runs on a single CPU and the benchmark executes for few seconds (less than 3 seconds as for the benchmarks with input S), the time spent on optimizing the code is an additive term of the total time of the dynamic compiler; hence the optimization time slows down the execution of ILDJIT.

An evaluation of the modularity design of ILDJIT should be discussed. Since ILDJIT is composed by external plugins, the maintenance and the extendibility of the dynamic compiler are achieved properties. On the other hand, this design leads to a slightly reduced efficiency comparing to a monolithic design.

As we can observe by the Figure 15, the time spent inside the compiler is negligible if ILDJIT does not optimize the code (ILDJIT0). For this reason, we have evaluated the loosed efficiency of ILDJIT for its modularity design, by including the optimization plugins inside the framework in order to avoid indirect function calls needed by a modularity design. We found that avoiding the indirect function calls (ILDJIT-Monolithic) the improved efficiency is totally negligible (between 1% and 2%). This result is due to the fact that the heavy computational part of the code optimization algorithms is inside the external plugins and not among their communications implemented through indirect calls.

In the following Sections, only ILDJIT-Auto is considered (by using ILDJIT name).

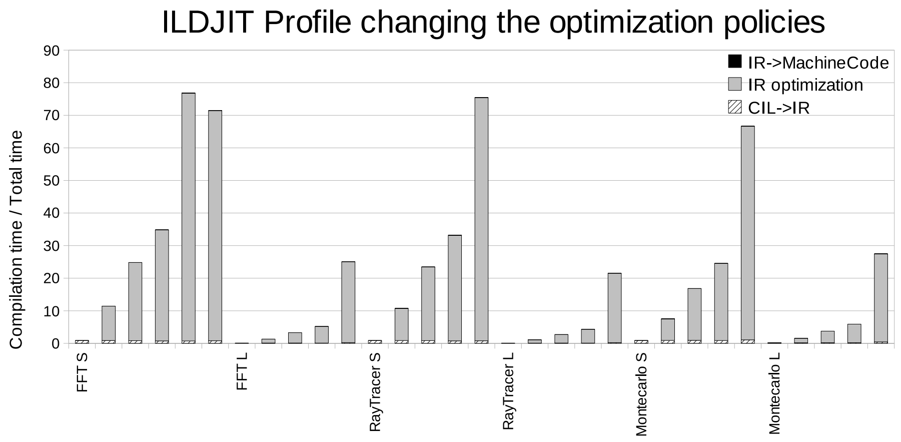


Figure 15. Time spent by ILDJIT over the compilation, optimization and execution phases using ILDJIT0 (1c), ILDJIT1 (2c), ILDJIT2 (3c), ILDJIT3 (4c), ILDJIT4 (5c) and ILDJIT-Auto (6c). The time spent executing the machine code is omitted in order to zoom into the compilation efforts. Only few benchmarks are provided because others expose the same pattern of results.

8.4. Impact of Parallel Compilation

To gauge the impact of pipeline compilation, we present a set of three experiments run on a four cores machine (Xeon E5335 at 2 GHz, 4MB of L2 cache, 8GB of ram), using 1,2 and 4 processors. With one core, the machine does not offer hardware parallelism and some overhead for the execution of ILDJIT is confirmed: the multi-threaded system needs time for synchronization between threads, without taking much benefit from parallelization.

Figure 16 shows the execution time of the VES running the benchmark suites on 2 and 4 cores versus 1 core. All reported timings are median elements over 20 runs on an otherwise idle machine. The execution time shown in Figure 16 diminishes by 22% on the average, using two cores and 44% using four cores. We observe that ILDJIT is able to achieve an already significant speedup using two cores. Moreover, using four cores, we are able to mask more than 90% of the compilation delay, thus achieving the mentioned speedup of 44%.

Figure 17 provides the information about the compilation efforts perceived by the user, which means the time spent by ILDJIT stalling the execution of the machine code in order to compile/optimize that code. As the first bar shows, on a single processor machine, our dynamic compiler spends most of the time executing the machine code and optimizing the IR methods. Using one core, the translation from IR to the machine code is less than 0.01%. Moreover, the compilation plus optimization phases account for less than 20% on benchmarks with input L and 69% on the same set of programs with input S (44% on average).

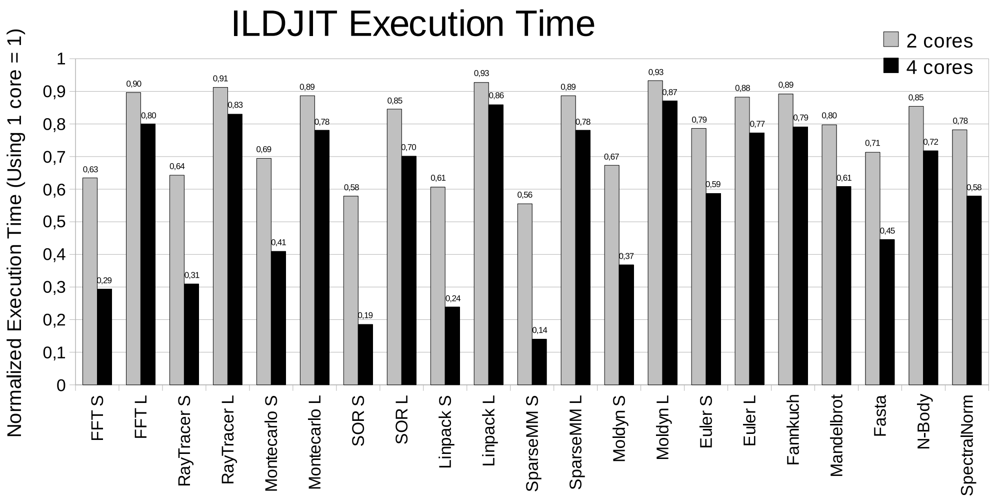


Figure 16. Total execution time spent by ILDJIT enabling or disabling CPUs; results are normalized to the execution time spent using a single CPU.

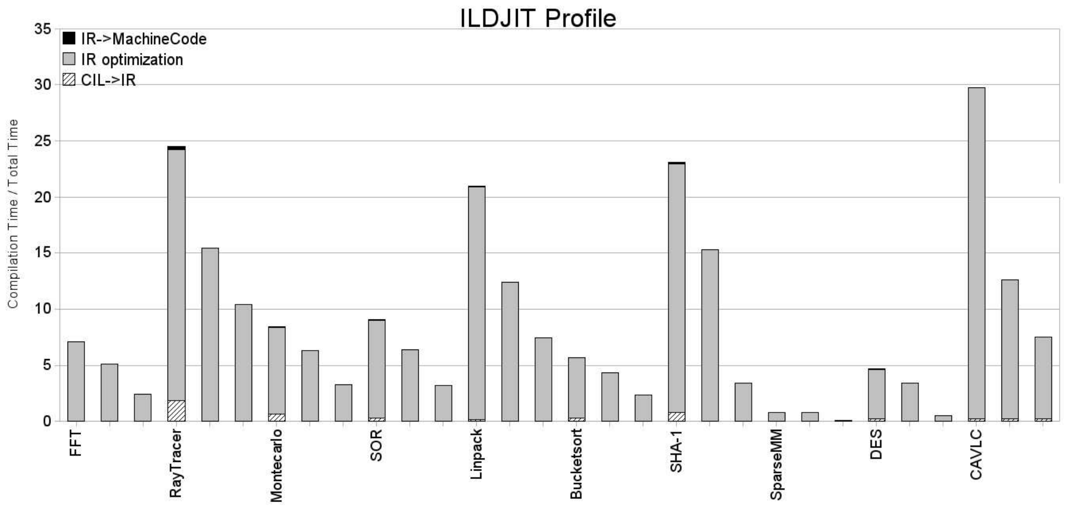


Figure 17. Time spent by ILDJIT over the compilation, optimization and execution phases using one (1c), two (2c) and four (4c) cores. The time spent executing the machine code is omitted in order to zoom into the compilation efforts.

The second bar of Figure 17 shows that two cores are sufficient to mask the translation time. More in details, the translation phases are less than 0.01% of the total execution time. The compilation plus the optimization phases are less than 2% for the benchmarks with size L and 58% for programs with size L (30% on average). It also show that for most of the time, ILDJIT executes the application code on benchmarks with size L (for more than 98% of the total execution time).

Finally, the last bar of Figure 17 shows that the time spent to optimize the code is almost entirely masked using 4 cores. More in details, the compilation and optimization phases are less than 3% of the total execution time.

From these measurements, we see that compiler parallelization is successful: the translation and optimization phases overlap the execution time and do not add to it.

Observing the single core case, the VES spends most of the time executing application code or, in some cases, optimizing it; it means that the translation from CIL to IR representation as well as the translation from IR to the machine code are almost negligible. More in detail, Figure 17 shows that the time spent on the second translation is totally insignificant (less than 0.1%). The explanation is that our design choice favors faster IR to native translation. The rationale is that continuous optimization can be employed to adapt dynamically the code, and requires IR to native, but not CIL to IR recompilation.

8.5. Comparison of VES Implementations

In this Section, we compare the performance of ILDJIT and its main open source competitors, Mono and Portable.NET. To decrease the bias due to the specific processor architecture, we take into consideration two different platforms: Intel x86 and ARM926. Here, we only consider single core platforms in order to show the ability of our dynamic compiler to generate efficient machine code for different underlying hardware independently from the effectiveness of the pipeline compilation and Dynamic Look Ahead techniques, and therefore to show that the baseline used for first part of the evaluation does not introduce false positive results as would happen if a low-quality baseline implementation was used.

The execution times on the Intel and ARM platforms are presented in Figure 18 and in Figure 19 respectively.

The results show that ILDJIT is faster than Mono and Portable.NET regardless of the hardware platform, when the input size is large enough. Specifically, on the proposed benchmark set, ILDJIT is able to outperform Mono by 32% on x86 for the large (L) input size. On the other hand, if we consider the small input size (S), Mono outperforms ILDJIT by 46% on x86.

The rational on these results is that ILDJIT is tuned for programs that runs at least for one minute on a general purpose platform like Intel. Currently ILDJIT targets medium/large benchmarks where the execution of the programs is not negligible.

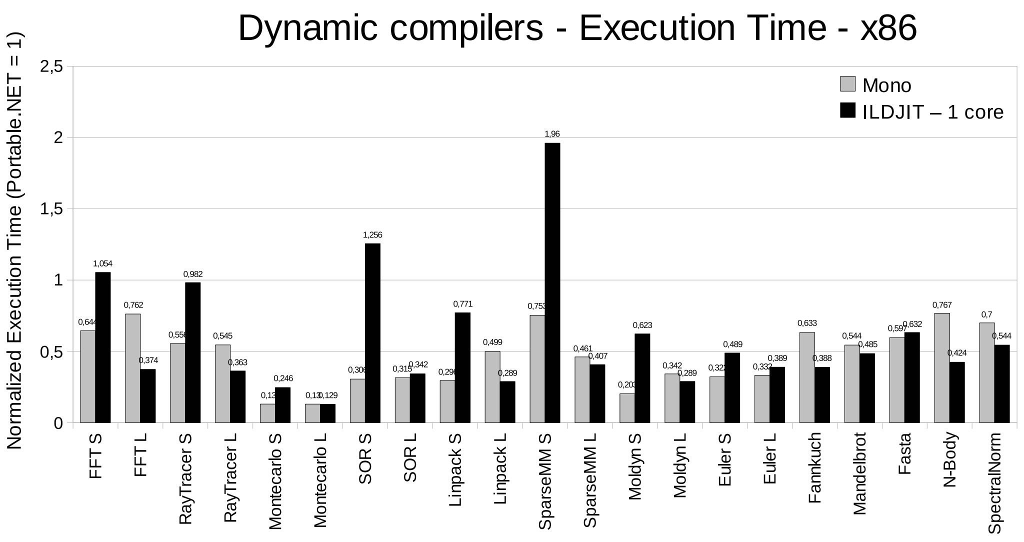


Figure 18. Total execution time on Intel x86 platform spent by ILDJIT, Mono and Portable.NET. Results are normalized to the execution time of Portable.NET, set to 1

8.6. Dynamic versus Static compilation

A comparison of the static and dynamic compilation is provided by using the CBench benchmark suite. Statically compiled code is generated by GCC; while dynamically compiled code is generated by ILDJIT. Results are taken by using Intel x86 as platform.

Figure 20 shows that ILDJIT runs faster than the not optimized static compiled code generated by GCC. Moreover, ILDJIT runs slower that the optimized version of the static compiled code.

9. Related Works

The present work is part of the long tradition of dynamic compilation. An historical perspective on Just-In-Time compilation can be found in [8], while Ahead-Of-Time compilation is discussed in [35]. In this Section, we will restrict the discussion to works closely related to the ILDJIT dynamic compiler.

Dynamic compilation is chiefly used in two different contexts: binary translation and bytecode compilation. Binary translation is used to support legacy binary software, providing compatibility between new architectures and existing ones [17, 21], or to provide support

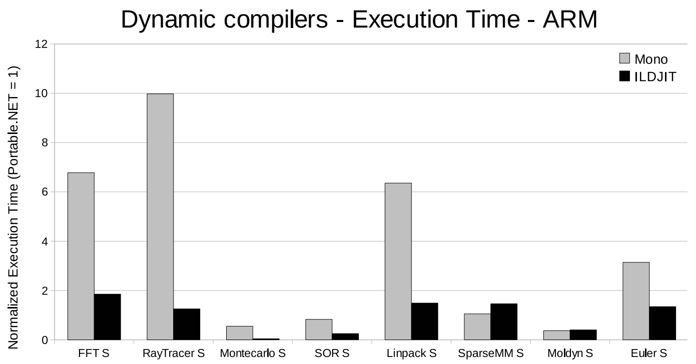


Figure 19. Total execution time on ARM platform spent by ILDJIT, Mono and Portable.NET. Results are normalized to the execution time of Portable.NET

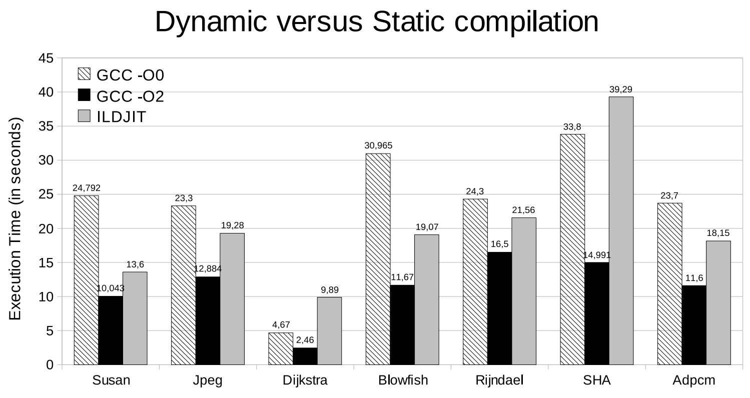


Figure 20. Total execution time of the static compiler GCC enabling or disabling the optimization (O0, O2) and ILDJIT.

to virtualization [4]. Bytecode compilation, on the other hand, is used to generate efficient code from portable bytecode languages. In general, bytecode compilation requires, in addition to translation itself, the full range of runtime support features that characterize a Virtual Machine.

A first work on the impact of separating compilation and optimization threads is [31]. There, the authors support the choice of increasing the share of computation power reserved to the compiler through an experimental campaign conducted on IBM's J9 [39] virtual machine, running several large Java benchmark suites on single-threaded and hyperthreaded hardware.

9.1. ECMA-335 Implementations

The most closely related works to our own are five other implementations of the CIL bytecode and CLI framework. Due to the considerable impact of the Microsoft ".NET" initiative, several projects aimed at VES implementation.

Microsoft itself released three implementations of VES: the .NET framework (for desktop environments), the Compact Framework (for embedded devices) and Rotor, a "shared-source" implementation. Mono is an open-source project led by Novell (formerly by Ximian) [16]. Portable.NET is an free software implementation of CLI by Southern Storm [36].

With respect to these VES implementations, ILDJIT has two main advantages: it is designed to exploit the multi-core hardware architectures as much as possible and to be easily extensible by exposing an internal software plugins framework, so switching between different implementations of the same module at runtime is possible. Moreover, the plugin framework allows ILDJIT to expose to the programmer policy modules to adaptively control optimization, thus providing a better platform for the exploration and evaluation of optimization policies. Section 8 provided a comparison of ILDJIT to the existing free/open source competitors (Portable.NET and Mono).

9.2. Java Virtual Machine Implementations

Much research on dynamic compilation has targeted the popular Java bytecode language. The *hotspot* detection technique [33] is used in Sun's Hotspot VM to switch from interpretation to just-in-time compilation after a method has been identified as a hotspot (i.e., it is frequently executed). This technique mitigates the cost of compiling rarely executed methods. However, interpretation can be very costly for large methods, even when executed only rarely. A similar technique is used in JIT-only virtual machines such as Jikes RVM [7], where adaptive optimization is based on execution profiling to detect hotspots, which will then be recompiled enabling more aggressive optimization.

Hotspot compilation could be combined with the compilation framework of ILDJIT, allowing small methods that are seldom executed to be interpreted rather than compiled. Adaptive optimization can also be implemented in our framework, leveraging the existing execution profiling facilities, to speed up the first execution of a method should the DLA technique fail to identify it early enough for all needed optimizations to be applied before its invocation.

In BEA's JRockit [2] virtual machine the methods are compiled without performing code optimizations for their first execution; the compilation are performed by the same thread used to execute the application code, but there is a parallel thread which has the task of sampling the execution, in order to trigger method recompilation, increasing the optimization level. In IBM's J9 [39] as well as in Intel's ORP [14] virtual machines there are parallel threads to perform code compilation and execution tasks. ILDJIT combines the exploitation of parallelism with a finer grained control of optimization, as well as support for multiple source programming languages via CIL bytecode.

A technique related to the Dynamic Lookahead Compilation used in ILDJIT is Background Compilation [30]. Optimization is performed on dedicated hardware, on the base of an off-line profiling phase. If a method still lies into the optimization queue at its invocation, lazy compilation is employed. The major advantage of DLA is that it does not require off-line profiling, while still being able to forecast the execution path.

When considering ILDJIT as an example of modular design of a Virtual Machine and dynamic compiler, a recent, comparable work is JnJVM [41], which is designed leveraging aspects [28] to obtain flexibility. In JnJVM case, however, the cost of flexibility is felt on the performance side, as JnJVM is competitive with respect to the Kaffe JVM [27], but not with other implementations of the JVM specification. On the other hand, ILDJIT, which implements a more complex specification (ECMA-335, e.g., supports unmanaged code in addition to managed code, which is not the case of the JVM specification), is competitive with Mono and Portable.NET, both of which are industrially developed.

Finally, industrial research in multi-threading on Virtual Machines leads to both multiple threads of compilation and multi-threading optimization. The Azul VM [25] employs both techniques to speed up application threads via speculation on the thread-safety of concurrent memory accesses and multiple compilation threads [31]. ILDJIT does not currently focus on application thread optimization, though its modularity allows for easy optimization of the thread subsystem: currently, ILDJIT supports two implementations of the threading subsystem, one based on the POSIX threads, designed to maximize portability, and one implementing thin locks [10] to keep the performances in line with the direct competitors.

10. Conclusions and future work

The work described is an important step towards exploitation of parallel dynamic compilation for parallel architectures such as the multi-core processors. The experiments reported, albeit initial, give evidence of the advantages in terms of reduction of initial delay and execution speed up. Moreover since ILDJIT is a young system, we expect forthcoming releases to perform significantly better. ILDJIT is designed on a pipeline model for the translation and execution of CIL programs, where each stage (CIL/IR translation, optimization, IR/native translation, and execution) can be performed on a different processor. This choice brings a great potential for continuous and phase-aware optimization in the domain of server applications, as well as fast reaction times and effective compilation on embedded multiprocessor systems.

Since ILDJIT exposes a plugin-based framework, it is customizable and extensible. Its flexibility makes it fit for different uses. Its primary application is currently as a research platform to investigate parallel compilation techniques, automatic parallelization and optimization heuristics. It is also used as a development and research platform to provide instruction virtualization in multimedia platforms as part of the OpenMediaPlatform [3] project.

Finally, its modularity makes ILDJIT suitable for educational purposes, as it shows how to practically build a working and industrially competitive compiler in a modular way. Moreover, the plugin optimization framework allows students to experiment with optimizations using a well defined interface. In this capacity, ILDJIT is used in teaching advanced compiler construction to graduate students at Politecnico di Milano.

Several interesting directions are open for future research. An important future direction for research is the study of scheduling policies for method optimization and the study of different threads schedule policies.

Dynamic optimizations that exploit profile information (like method inlining, method specialization, ecc...) in order to adapt the code at runtime is an ongoing work.

Inter-procedure and inter-thread code optimizations and memory alias analyzers can be applied in ILDJIT; hence they can be taken into consideration in a static and dynamic scenario.

Since ILDJIT is composed by external plugins, the impact of different garbage collector algorithms can be studied within this framework. Finally, machine dependent optimizations could be explored, e.g. by replacing the current linear scan register allocation with more time-consuming algorithms such as graph coloring or puzzle-solving [34].

Technical developments towards full compliance with the ECMA-335 standard include support for more source languages beyond C and C# and support of non-functional parts of the specification (e.g., security).

Acknowledgements

This is a large multi-year project and several individuals have contributed or are working on it, within the Formal Language and Compiler Group. In particular we would like to credit Michele Tartara for ARM support, and Ettore Speziale for his contribution to the support of the CIL delegates. Marco Cornero and Erven Rohou of ST Microelectronics have stimulated our commitment to free software development of dynamic compilation platforms.

REFERENCES

1. The Computer Language Benchmarks Game. http://shootout.alioth.debian.org.
2. Bea jrockit: Java for the enterprise technical white paper, 2006.
3. Open Media Platform (OMP) European project. http://www.openmediaplatform.eu, 2008.
4. Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.
5. Giovanni Agosta, Stefano Crespi Reghizzi, and Gabriele Svelto. Jelatine: a virtual machine for small embedded systems. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 170–177, New York, NY, USA, 2006. ACM.
6. Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the FANNKUCH benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, 1994.
7. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

8. John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.
9. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
10. David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM.
11. Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.
12. Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. Benchmarking Java Grande applications. In *in Proceedings of the Second International Conference on The Practical Applications of Java*, pages 63–73, 2000.
13. Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi-Reghizzi. Dynamic Look Ahead Compilation: A Technique to Hide JIT Compilation Latencies in Multicore Environment. In *Proceedings of the 18th International Conference on Compiler Construction (CC 2009)*, pages 220–235, March 2009.
14. Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):617–637, 2005.
15. Roberto Costa, Andrea Ornstein, and Erven Rohou. http://gcc.gnu.org/projects/cli.html. GCC4NET.
16. Miguel de Icaza, Paolo Molaro, and Dietmar Maurer. http://www.go-mono.com/docs. Mono documentation.
17. James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.
18. Robert L. Devaney and L. Keen. *Chaos and Fractals: The Mathematics behind the Computer Graphics.* American Mathematical Society, Boston, MA, USA, 1997.
19. Evelyn Duesterwald. Dynamic Compilation. In Y.N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook — Optimizations and Machine Code Generation*, pages 739–761. CRC Press, 2003.
20. Eric Dussaud, Chris Husband, Hoang Nguyen, Dan Reynolds, and Christiaan Stolk. Hundred Digit Challenge Solutions. Technical report, Department of Computational and Applied Mathematics, Rice University, Houston, TX, USA, May 2002. See [**?**, **?**, **?**].
21. Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.
22. ECMA. *ECMA-335: Common Language Infrastructure (CLI).* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fourth edition, June 2006.
23. Grigori Fursin. Collective Tuning Initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009.
24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, January 1995.
25. Brian Goetz. Optimistic thread concurrency. Technical Report AWP-011-010, Azul Systems, Inc., January 2006.
26. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.
27. Kaffe. http://www.kaffe.org.
28. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220–242, June 1997.
29. Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.
30. Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.
31. Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic compilation: the benefits of early investing. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 94–104, New York, NY, USA, 2007. ACM.
32. P. A. Lee. Exception Handling in C Programs. *Softw., Pract. Exper.*, 13(5):389–405, 1983.

33. Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.
34. Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226, 2008.
35. T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler. In *Proc. of the Third Conference on Object-Oriented Technologies and Systems*, Jun 1997.
36. Southern Storm Software. http://www.dotgnu.org/. DotGNU Portable.NET project.
37. W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1.* Pearson Education, 2003.
38. Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, 2006.
39. Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society.
40. Michele Tartara, Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. Just-In-Time compilation on ARM processors. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 70–73, New York, NY, USA, 2009. ACM.
41. Gaël Thomas, Nicolas Geoffray, Charles Clément, and Bertil Folliot. Designing highly flexible virtual machines: the jnjvm experience. *Softw. Pract. Exper.*, 38(15):1643–1675, 2008.
42. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.