A parallel dynamic compiler for CIL bytecode

Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi
Politecnico di Milano
{campanoni,agosta,crespi}@elet.polimi.it

Abstract

Multi-core technology is being employed in most recent high-performance architectures. Such architectures need specifically designed multi-threaded software to exploit all the potentialities of their hardware parallelism.

At the same time, object code virtualization technologies are achieving a growing popularity, as they allow higher levels of software portability and reuse.

Thus, a virtual execution environment running on a multi-core processor has to run complex, high-level applications and to exploit as much as possible the underlying parallel hardware. We propose an approach that leverages on CMP features to expose a novel pipeline synchronization model for the internal threads of the dynamic compiler.

Thanks to compilation latency masking effect of the pipeline organization, our dynamic compiler, ILDJIT, is able to achieve significant speedups (26% on average) with respect to the baseline, when the underlying hardware exposes at least two cores.

Keywords: dynamic compilation, parallel virtual machine, Virtual Execution System.

1 Introduction

This paper reports the project of a new dynamic translator and optimizer (DTO) [1] for the widespread Common Intermediate Language (CIL), the directly interpretable representation of the Common Language Infrastructure (CLI), standardized as ECMA 335 and ISO/IEC 23271:2006 [8].

The current name of the project is ILDJIT. It is released under GPL license through Sourceforge. Most existing DTOs have been designed for single-processor machines and their performances when moving to multi-processor or multi-core machines are likely to be suboptimal. Our project from the very start 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 DTO 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. 2007 [12]

> *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 DTO 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.

Third, the DTO 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 DTO modules are implemented as plugins.

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 early experiments show that the performance of applications compiled by our DTO are comparable to some of the best known dynamic compilers, on a single processor machine, and superior on multi processor platforms.

As we have implemented just a few basic optimization processes in our current release, we expect much improvements will come as we enrich the optimization

[1] We refer to the classification and terminology for dynamic compilers proposed by Rau [15] and Duesterwald [7]

set and we gain experience on the policy for triggering optimizations.

In this paper we report on single-thread applications, as the support for multiple C# and Java threads is under development. The paper is organized as follows. In Section 2 we outline the execution model of ILDJIT dynamic compiler. In Section 3 we outline the DTO architecture, the intermediate representation and explain the parallel organisation of the compiler. Some finely tuned modules of the compiler are also described. In Section 4 we report the experiments and how they allowed us to tune performances and improve on the overall DTO structure. Some benchmarks are compared with other CIL systems. The conclusion lists on going developments and future plans.

2 Execution model

ILDJIT implements the Virtual Execution System (VES) leveraging on a Just-In-Time compiler for obvious performance reasons. The primary task is to translate each piece of CIL bytecode to a semantically equivalent target code to be directly executed by the hardware; ILDJIT adopts an intermediate representation called IR to support this translation.

2.1 Translation unit

Choosing the correct granularity for the translation process is especially important in a dynamic compiler [7]. A larger translation unit may provide additional optimisation 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 optimisation, 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. This is not needed if the unit is composed of one or more functions, since the function call boundary naturally defines a state to be preserved across the call (parameters and return values), while most of the local state can be destroyed (local variables of the callee). Other advantages that make the CIL method a good candidate for the role of translation unit are specific to the CLI. 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.

Therefore ILDJIT uses the method as its translation unit. The choice of larger units is possible, and can be effectively achieved by method inline pass plugin.

2.2 Intermediate representation

ILDJIT has to optimize application code over different processing units (PU), which communicate using communication channels like shared memory or TCP/IP.

To this end, ILDJIT must sometimes move code across different PUs which may be connected through slow communication channels. Clearly, communication costs should be minimized. Next we explicit the rationale for adopting IR as internal program representation shared on these communication channels.

CIL is not a good candidate for the internal representation of methods because the information needed to completely describe a method is spread on the metadata of its CIL container which also describes many other aspects of the entire program. Moreover, CIL provides a very compact representation of an entire program, but forces the compiler to access this representation in many unrelated points to collect the information needed by a single method. Nor would any machine code be a good candidate, as it would be too hardware specific. The translation cost for mapping a machine code onto a very different one would be unwarranted. It is well known from static compilation experience that many optimizing transformations are better performed on an intermediate representation than on machine code; the same reasons apply here.

We have therefore designed our own IR language, with two aims: first, to provide a very compact representation of individual methods by removing the dependences on CIL metadata streams; second, to offer a machine-neutral, register-based language, such that each construct has a clear and simple meaning. This has the advantage that optimizing transformations that rewrite IR are easy to specify and implement.

Translation is therefore split in two phases: first the CIL fragment is translated into IR, then IR is translated into the target machine code. Consider a method composing the CIL program given as input to the ILDJIT compiler. The method can be in the following non-overlapping *translation states*: CIL, IR, MACHINECODE and EXECUTABLE. A method is in *CIL state*, if it is present only in the CIL language; otherwise, if a method is in the *IR state*, it is present both in CIL language and in IR. In *MACHINECODE state*, a method is present in CIL, IR, and in the target machine code. Finally in EXECUTABLE state, the method is present in CIL, IR, and machine code and all the static memory used by it is allocated and initialised. To change its translation state, a method traverses the software pipeline next described in 2.3.

2.3 Compilation pipeline

Translations, optimization and execution of CIL code is managed by an internal software pipeline, designed to exploit hardware parallelism at various levels. To this end, compilation and execution phases are performed in parallel. Moreover ILDJIT exploits pipeline parallelism, to add another dimension of parallelism between compilation, optimisations, and execution phases.

In traditional Just-In-Time compilers, when execution jumps to a method not yet available in machine code, it pauses and translates it. Our dynamic compiler, given sufficient hardware resources, can produce methods in machine code before execution of the CIL program asks for them. In this case, program execution does not need to be paused to switch to compilation; the execution profile matches that of a statically compiled program in the optimal case.

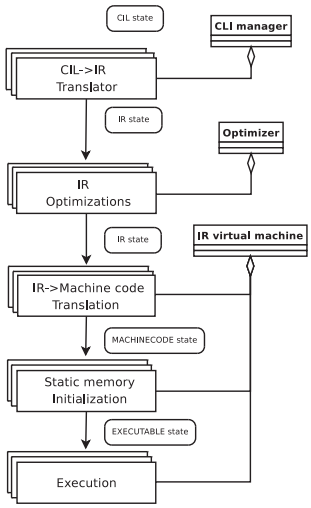

Figure 1: The translation pipeline model: on the left, the translation pipeline stages; on the right, the modules that implement the various stages; in the rounded boxes the state of the method code at the beginning of each stage

The pipeline model exposes five stages as shown in Figure 1 and explained in details in Section 3. All stages can be parallelized, if hardware resources are available. Each pipeline stage can be performed by several parallel threads, possibly running on different PUs', 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 pipeline model is implemented by the Pipeliner module (see Section 3.4).

2.4 Linking

Since the entire program is not translated to target code at once, every dynamic compiler has the following linking problem: how to handle invocations of methods not yet translated. In this respect our DTO is rather classical. ILDJIT redirects invocations to the internal Execution engine module (see Section 3.3.2), which then calls the right internal modules responsible for translation; after that, it can re-link the freshly translated method to the rest of the program. This redirection mechanism is known as a *trampoline*. ILDJIT resolves the linking problem by a lazy compilation [11].

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 each method (as opposed to a single dispatch function in static compilation) comes from this principle.

2.5 Precompilation

As explained above, ILDJIT can prefetch CIL methods to translate and optimize them before their execution is demanded by the application. A critical decision is which method to compile ahead of time.

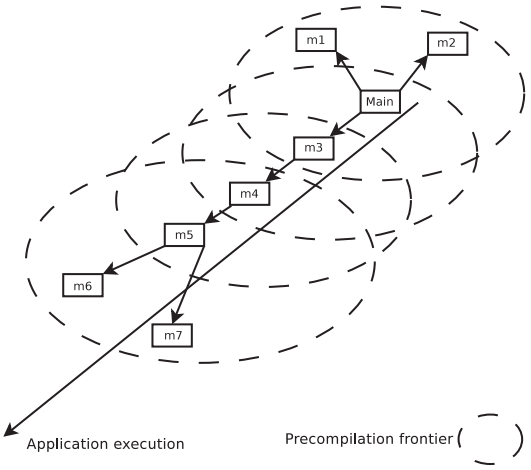

Figure 2: Adaptation of the precompilation frontier to the execution of the application code; the executing methods are in order: Main, m3, m4, m5; the threshold D is 2, constant.

Our simple yet effective selection criterion is based on *method distance*. We define the distance of a method M_i from the method currently in execution, as the shortest path over the call graph between M_i and the latter method.

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 not only are in no hurry for being translated, but have low probability of being requested in the future, because the call graph chain ignores conditions. The threshold D is adjusted at runtime depending on available free PUs'.

We define the *precompilation frontier* as the portion of the call graph of the application code, which has the following meaning: it includes the next candidate methods for compilation and optimization tasks. The precompilation frontier is a dynamic set, moving along with the executing method at a distance depending on the threshold D and available system resources. An example of precompilation frontier and of its adaptation to the evolution of the execution of the application is shown in Figure 2; in this example the system resources available are considered constant as the threshold D.

As more PUs become available, the precompilation frontier widens; a large frontier means ILDJIT compiles ahead of time many methods and then the probability of spending time inside the trampolines decreases. An example of precompilation frontier and of its adaptation to the system resources available is shown in Figure 3.

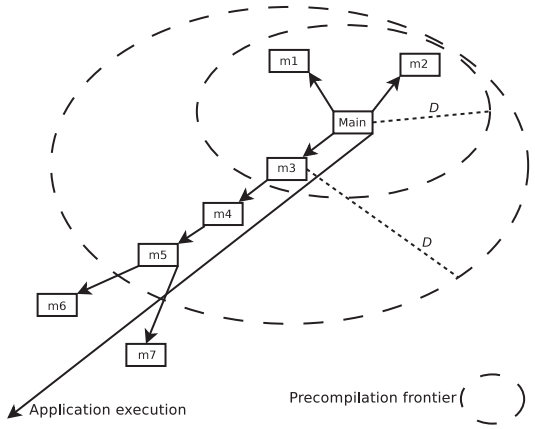

Figure 3: Precompilation frontier with adaptive threshold D; D evolves from 2 to 3; the executing methods are in order: Main, m3.

2.6 Optimizations

ILDJIT allows optimizations both at IR and target code level. The rational is based on the observation that all the semantic information of the source program is available, higher-level transformation are easier to apply. For instance, array references are clearly distinguishable, instead of being a sequence of low-level address calculations [4]. An instance of transformation that is useful at each levels are the loop-invariant code motion; it can be applied at a IR level to expressions or at a target code level to address computations. The latter is particularly relevant when indexing multidimensional arrays [4].

Since different algorithms for code optimization use particular features of the underlying hardware, ILDJIT can optimise the code that is going to be executed at the IR level, making a translation from and to the IR language, and at the target machine code level, making a translation from and to the target machine code.

IR to IR optimizers run as independent threads possibly on different PUs', or even on different machines connected by an IP network.

The decisions about when, where and how much to optimise each CIL method are taken based on the method distance concept (see Section 2.5).

In the future we expect to use profiling for identifying methods containing hot spots, as the first candidates for aggressive optimization.

3 Software architecture

The execution model described in Section 2 and shown in Figure 1 is implemented in the ILDJIT modular software architecture, composed by the following main blocks:

Bootstrapper This is the first module executed at ILDJIT system startup time; it provides initialisation services for all system components.

Pipeliner implements and manages the pipeline needed for translations, optimisations, and execution of CIL bytecode.

CLI Manager provides the functionalities needed to implement the CLI architecture, including the translator of the CIL bytecode, the layout manager for the CIL objects, the CIL core libraries and the loader/decoder of the CIL files.

Optimizer implements the optimising transformations of IR methods.

IR virtual machine It is the core module, responsible for translating IR code to machine code, and for running the target code.

Garbage collector All memory used by ILDJIT , both in execution and compilation phases, is allocated and managed by this component.

Threads manager This module manages the CIL threads by grouping and scheduling them on PUs.

Policies This module implements various policies that ILDJIT has to use inside dynamic compilation phases (e.g. optimization policy).

Profiler Profiling functionalities for the various modules of ILDJIT dynamic compiler are provided by this module.

Tools All generic tools exploited by the ILDJIT dynamic compiler are implemented inside this module.

For a complete description see [1]. The dependences between the ILDJIT's modules are shown in Figure 4. The dependences to the Tools, to the Profiler and to the Garbage collector are left implicit.

Each main block is further modularised, though different approaches to modularisation have been chosen, depending on the typical use of a module:

Dynamically loaded shared library This choice gives the highest degree of flexibility, allowing the module to be loaded at runtime, if available. It is therefore employed for all components that can be freely replaced, added or removed from the system. Our dynamic compiler is designed to be easily extendible: the use of dynamically loaded shared libraries allow the implementation of a plugin framework, making the dynamic compiler customisable for a specific application domain [14] (e.g. for multimedia component).

Statically loaded shared library Some core components need to be present at all times in the system, yet they are used by several subsystems that can run on different processors. In this case, statically loaded shared libraries allow a good degree of flexibility, while removing unnecessary loading overheads [14].

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

The rest of this Section describes in greater details the most significant modules.

3.1 CLI manager

The CLI manager has the task of managing the CIL bytecode as specified by the ECMA-335 standard [8]; for this task it has to: load and decode the CIL files; translate CIL methods to our IR language; layout the

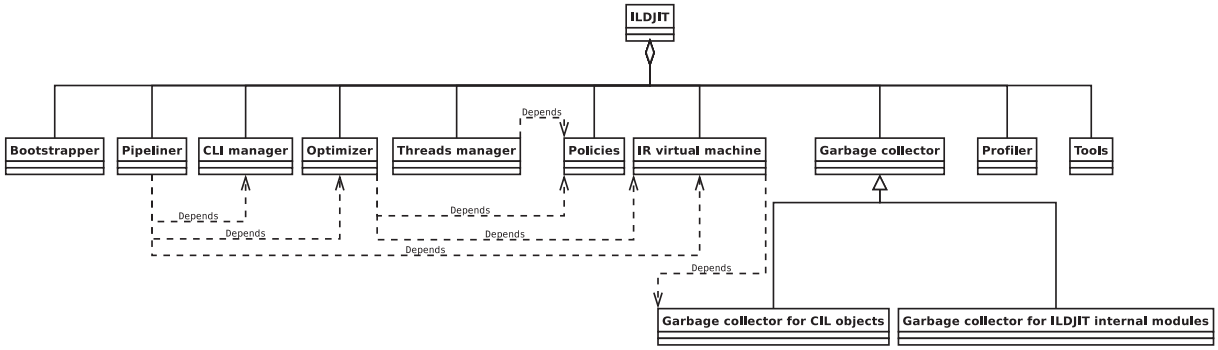

Figure 4: Class diagram of ILDJIT system

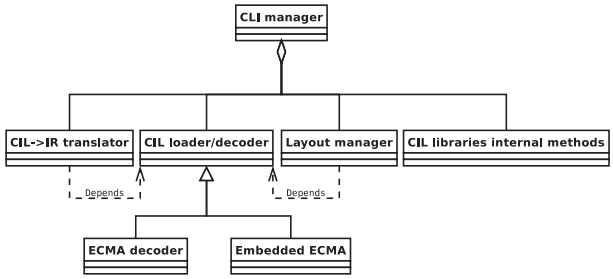

Figure 5: Class diagram of the CLI manager module

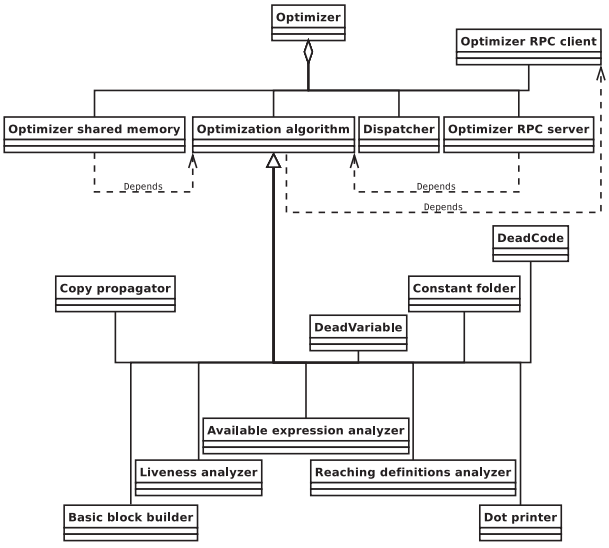

Figure 6: Class diagram of Optimizer module

instances of the CIL classes; and provide the implementations of the various internal calls of the C# base class library. The software architecture of the CLI manager is shown in Figure 5.

The loading and decoding tasks of the CIL files are performed using external modules (plugins) since these tasks can be implemented in many different ways with different pros and cons. ILDJIT currently includes two such plugins: the ECMA decoder, and the Embedded ECMA decoder. The first plugin uses a full caching policy for the decoded information, thus decoding every file only once. The embedded ECMA decoder, on the other hand, does the decoding on demand, and only keeps a limited cache. The two plugins thus target different platform types: the first is suitable for desktop/high-end systems, while the latter is suitable for memory-constrained embedded systems. As opposite, the ECMA plugin needs to decode and load each CIL file only once because it keeps every information in memory.

The translation of CIL methods to IR is performed by an internal module that converts the stack based CIL language to the register based IR language.

The layout manager computes and caches the memory size of each object type, and provides this information to the runtime exception engine.

The ECMA-335 standard describes the CIL core libraries usable by the CIL programs to perform their tasks. These libraries for the most part are implemented in C# language as external libraries. ILDJIT currently uses the C# implementation of these libraries developed within the Portable.NET project [16]. Some CIL methods (e.g. the methods in the System.Reflection namespace) have to be provided by the virtual machine. We have implemented these methods natively for performance reasons.

3.2 Optimizer

This module implements the optimisation phase for IR methods composing the application. The software structure of this module is shown in Figure 6.

In traditional dynamic compilers, the time spent for code optimisation adds to the total time needed for application program execution; this happens because optimisation time does not overlap with translation and execution time. To avoid lengthy optimization of rarely executed methods, a dynamic compiler needs an *optimization policy* to decide the best level of optimization of each method undergoing translation. In ILDJIT too the problem exists but is less critical, since the time spent by optimisers can be partially overlapped with the translation and execution time, because such activities may run on different PUs.

We have chosen to encapsulate the optimization policy decision inside the policy module; currently we have provisions for eight optimisation levels: from

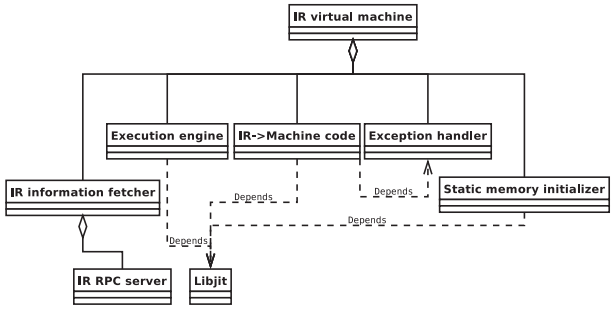

Figure 7: Class diagram of the IR virtual machine

standard optimization like constant propagation, constant folding and dead-code elimination to the last level, where all the optimisation algorithms implemented inside ILDJIT system are used. The current policy works in the following way: if the execution of the application code is waiting the translation of a method (inside a trampoline), the method is placed inside the first optimisation level, otherwise there is a heuristic to choose the right level, which is based on a back-propagation neural network. The network was trained off-line using several benchmarks.

Since the variety and number of possible optimisers are large and dependent on application and on target hardware, we have chosen to implement them by external plugins. The current release contains and the basic code transformations [3] [20] and the basic data flow analysis algorithms.

The IR methods to be optimised may reside on different machines connected by an IP network or on different PUs' connected by shared memories; in the former case we use the ONC RPC model for communication, [14] [18], in the latter we use simple, user-space, shared memories. The choice of the Inter Process Communication (IPC) protocol is automatically taken by the policy module of ILDJIT. Currently we have implemented and are evaluating the following policy. If, when the choice is due, any PU is idle, it is assigned to run the IR optimizer using shared memory as communication channel. Otherwise ILDJIT uses the remote machine for the optimization task through ONC RPC communication channel.

3.3 IR virtual machine

The IR virtual machine implements the entire virtual machine for our IR language. Due to its complexity, the module is further divided into five submodules, as shown in Figure 7: translating code from IR to machine language; managing the integration between different translation units; managing exceptional behaviour in the executed code; initialising the static memory; fetching information about IR data types and/or IR methods.

3.3.1 IR - Machine code translator

The *IR→Machine code translator* submodule translates a method from its IR representation to the equivalent target machine code. This submodule is called by the Pipeliner (see Section 3.4) module according to its policy.

The virtual machine relies on Libjit [17], a JIT compilation library, currently available for x86 and Alpha, to provide the final steps of translation from IR to machine code. IR was designed to be compiled as quickly as possible using Libjit, or similar macroprocessing tools, thereby minimising the overhead of translation to the target machine. Additional targets can be supported by re-targeting Libjit. This module converts the IR method to the format required by Libjit library, then exploits the API of this library to generate machine code.

All the classic optimizations available using the machine code [4] are performed on this phase.

3.3.2 Execution engine

The Execution engine submodule is in charge of executing the machine code of the application. It ensures that the execution of a method is stopped if it jumps to another method not present in machine code. It implements this functionality by trampolines (as explained in Section 2.4). Inside trampolines, the Execution engine calls the Pipeliner to insert into the software pipeline the current method, then waits till it exits from this pipe, thus indicating its readiness for execution.

As already explained, translation and the execution may run in parallel.

3.3.3 Exception manager

The exception manager module implements the mechanism to handle the exceptions thrown by method execution [9]. The implementation is based on long jump machine code instructions.

3.3.4 IR information fetcher

This module provides functionality of fetching information about IR data types and/or IR methods. There is a necessity of fetching these kind of information for the IR methods optimizations; for instance, an in-lining optimization needs information about another IR method rather than the current in analysis.

This module includes an RPC server because the Optimizer module (see Section 3.2) can have necessity to fetch information on IR data types and/or it can have necessity to fetch different IR methods (for example for in-lining optimization purpose).

3.3.5 Static memory initializer

This module dispatches the code needed to initialise the static memory of the application code. For example, if a program uses a static memory, then there can be a CIL method for its initialisation (usually called *cctor*) which it has to be called before the its first memory use.

In ILDJIT static memory is initialized just before the first use (lazy policy). The translation of a CIL method M_i, which is the first one that dynamically uses a static memory element, produces a list of other CIL methods, which are needed for the initialisation of the element. These methods may be then translated to machine code and executed by this module just before the execution of M_i.

Notice that the software pipeline (see Section 2.3) ensures that static memory is always initialised before its first use.

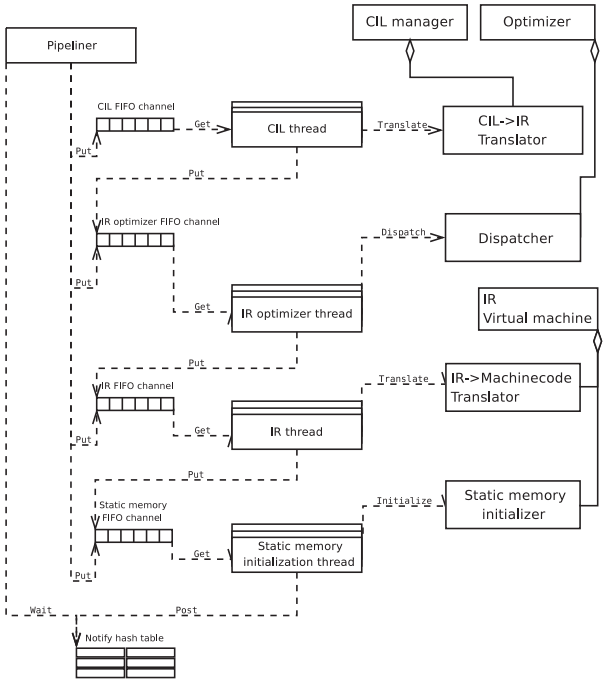

Figure 8: The Pipeliner's structure: in the middle four groups of boxes represent four groups of threads, each one assigned to a compilation task; on the right, the modules that implement the various compilation tasks

.

3.4 Pipeliner

The Pipeliner internal module implements the pipeline model described in Section 2. The model is organized by the threads depicted in Figure 8. As the organization is rather complex, we first describe it in four subsections: in Section 3.4.1 the algorithm for balancing the threads over the pipeline stages; in Section 3.4.2 the communication policy between the stages; in Section 3.4.3 how the Pipeliner inserts a new method into the pipeline; finally in Section 3.4.4 the calling convention for the Pipeliner module.

3.4.1 Threads balancing

The four groups of threads respectively implement the first four stages of the pipeline (Figure 1), as shown in Figure 8. At any time, the number of threads composing each stage of the pipeline is adaptively chosen, based on the current machine load where ILDJIT is running, and on the pending compilation workload to be supported. For each stage of the pipeline (except stage 4 static memory initialisation), the number of threads range between a minimum and a maximum; such values are set at bootstrap time. Then the Pipeliner dynamically adapts the number of threads for a stage using the histeretic model shown in Figure 9. The number of threads for stage $i, 1 \leq i \leq 3$ essentially depends on how many methods are present in the stages from 1 to i.

On the other hand Stage 4 of the pipeline behaves differently from the others, because limiting the number of threads may cause a deadlock. A situation that

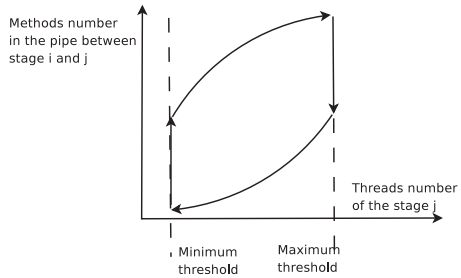

Figure 9: Histeretic pattern used for adaptation of thread numbers.

causes a deadlock is the following: we consider a maximum number of threads S for stage 4, we suppose there exist $S+1$ CIL methods which initialise the static memory and we suppose the call graph of the S CIL methods is a linear chain. To better explain why a deadlock exist, here is reported the actions performed by the system: ILDJIT starts putting the method M_1 into the top of the pipeline and then repeats the following actions S times:

- the method M_i goes through the pipeline till the static memory initialization phase;
- a thread of the fourth phase of the pipeline is allocated for executing all the methods needed to initialize the static memory used by M_i;
- M_i uses a static memory which impose to execute the method M_{i+1} before its execution;
- the method M_{i+1} is push on top of the pipeline synchronously (see Section 3.4.4);

When ILDJIT put on top of the pipe the method M_{S+1}, then there is no more thread at the fourth stage.

As the above example shows, we cannot bind the number of threads of the last stage of the software pipeline; for this reason we allow the number of these threads increase as much as the compilation workload request.

3.4.2 Inter-thread communication

Communication between the stages of the pipeline is implemented by means of FIFO software pipes residing in shared memory. Note the FIFO policy does not guarantee that the methods enter and exit the pipe in the same order. This out-of-order phenomenon is however irrelevant, since the Pipeliner does not make any assumption on the ordering of the methods under translation.

3.4.3 New method insertion

The input to the Pipeliner is a method composing the application code.

The method should enter the pipe at a stage, depending on its translation state (introduced in Section 2): for example, if the method is already in IRCODE state, the first stage is skipped. A method in CILCODE state has been loaded, but not translated to IR. Therefore, it enters the FIFO channel for the CIL threads. A method in state IRCODE has

been translated to IR, but not to machine code, and a method in MACHINECODE state is ready for execution. Methods in IRCODE state enter the FIFO channel for the IR optimizer threads. Methods in MACHINECODE state enter the FIFO channel for the Static memory initialisation threads. Methods in EXECUTABLE state bypass the pipe to its end.

3.4.4 Calling convention

To translate a method, the Pipeliner can be called synchronously or asynchronously; in the first case, control returns to the callee only when the method is ready for execution. In the second case, the Pipeliner puts the method on the right pipe, as explained in Section 3.4.3, and returns immediately to the callee. Synchronous calls are performed by the Execution engine module of the IR Virtual Machine 3.3.2 in order to translate methods that must be immediately executed. Asynchronous calls are performed by the CIL→IR translator 3.1 for methods that can bear a translation delay; if there are enough computational resources, such methods too will be translated, since they are likely to be needed soon (see Section 2.5).

3.5 Garbage collector

The garbage collector module provides a range of memory management functionalities. The simplest service is to allocate portions of memory to programs at their request, and to free them for reuse when no longer needed. A portion of memory can be declared useless automatically or explicitly. Memory requested to the garbage collector belongs to two sets: first, the memory requested for the internal modules of the dynamic compiler; second, the memory requested by the execution of the application program, which therefore contains only instances of CIL classes. The two memory sets expose different characteristics. A chief difference is that, while the memory used for the internal modules of the dynamic compiler can be freed explicitly by ILDJIT, the memory used for CIL objects has to be automatically freed, without any explicit notification coming from execution of the application. Such differences, as well as other ones, motivate our split of the memory manager into two tasks, each one applying different algorithms tuned to the different requirements.

Since a garbage collector can be implemented in many different ways, we have implemented this functionality by external modules.

An interface called *GarbageCollectorInterface* exposes the methods that each garbage collection plugin has to implement; such plugin can be used for both memory set, but two heaps exists in any case for the two memory sets. The interactions between the IR virtual machine and the garbage collector are bidirectional. The garbage collector could have necessity of calling the VM for the following reasons:

- to request the computation of the root set [21];
- to retrieve the list of objects referenced by the one given as input;
- to check if an object can be moved from a memory location to another one (there can be this kind of constraints on some objects [8]);
- to call finalizer methods for the objects marked as garbage by the garbage collector itself.

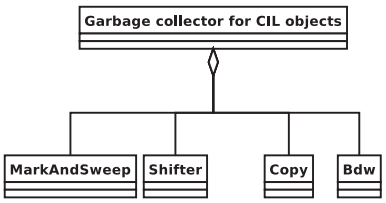

Figure 10: Class diagram of the Garbage collector module for CIL objects

These four tasks are demanded to the VM because the garbage collector does not known:

- the stack frame of the methods to compute the root set; in fact only the IR virtual machine knows this kind of information (see Section 3.3);
- the layout of the objects to compute the list of objects reachable at one step by a generic object; in fact only the CLI manager knows this information (see Section 3.1);
- how-to translate and execute a CIL method, in fact the execution of a method can be performed only by the IR virtual machine (see Section 3.3).

All the memory references are given to the garbage collector by their references because some collection algorithms needs to move objects over the heap automatically managed [21].

Because memory management greatly affects performances, we have four garbage collectors, suited to different situations, as shown in Figure 10. The first three are our implementations. *MarkAndSweep* implements the mark and sweep algorithm [21]; *Shifter* is our implementation of the mark compact garbage collector algorithm [21]; *Copy* implements the copying garbage collector algorithm [21]; finally, *Bdw* is a wrapper for the garbage collector of Boehm, Demers and Weiser [5]. The default configuration uses Bdw to manage the memory needed by the internal modules of the dynamic compiler and the Shifter for the instances of CIL classes.

4 Experimental evaluation

In this section we describe the experimental evaluation of ILDJIT dynamic compiler giving the following information:

- The execution times spent by ILDJIT using one or two hardware cores.
- The profiling information of ILDJIT for each compilation task; this information are needed to show where the system spends most of its time.

For this purpose we have chosen a set of single-thread benchmarks written in CIL bytecode, because

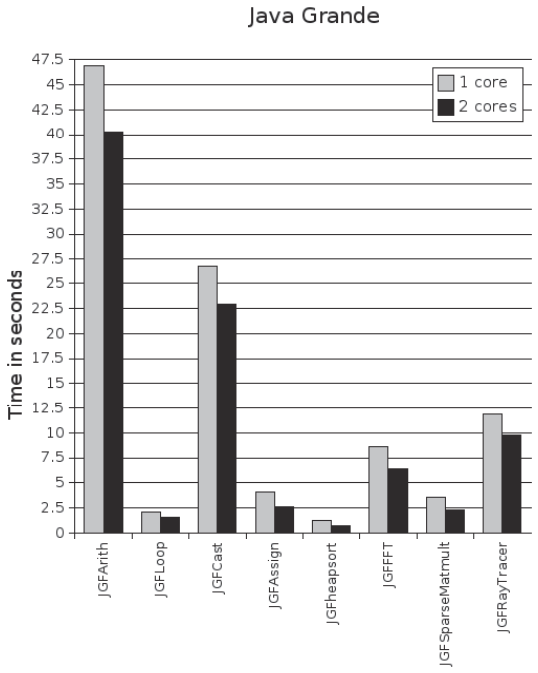

Figure 11: Total execution time spent by ILDJIT to run the Java Grande benchmark suite; the speedup in the 2 cores hardware configuration is due to the compilation phase overlapping done by the VM

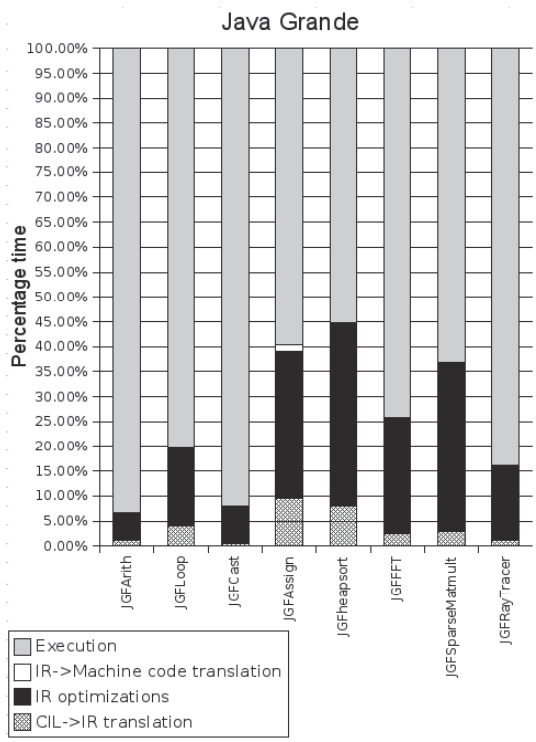

Figure 12: Time spent by ILDJIT over the compilation, optimization and execution phases using a single core; the translation from IR to machine code is less than 0.1%

our primary goal is to show the speedup that has been obtained implementing the VES virtual machine as a multi-threads system organized in pipeline.

We have chosen the benchmark suite Java Grande, which is a set of programs written using the C# language (hence translatable to CIL bytecode), part of the *Java Grande Forum Benchmark Suite* [13] [10].

4.1 Experimental Platforms

We performed our experiments on the same two cores machine, enabling or disabling a PU: Intel Core 2 Duo at 2.4 GHz, 2MB of L2 cache, 1GB of ram.

For the first experiment we use only one core of the machine. Since in this case the machine does not offer hardware parallelism, some little overhead for the execution of ILDJIT is confirmed as expected: in fact a multi-threads system needs time for synchronization between threads, without taking much benefit from parallelization.

For the second experiment we use two cores, which produce a significant speedup of our dynamic compiler.

4.2 Experimental results

Figure 11 shows the execution time of ILDJIT running the benchmark suite. All reported timings are median elements over 20 runs on an otherwise idle machine. The execution time shown in Figure 11 diminishes by 26.634% on the average, using two cores.

As Figure 12 shows, on a single processor machine ILDJIT spends most of time executing the target machine code and optimising the IR methods. Figure 13 shows that two cores are sufficient to overlap the trans-

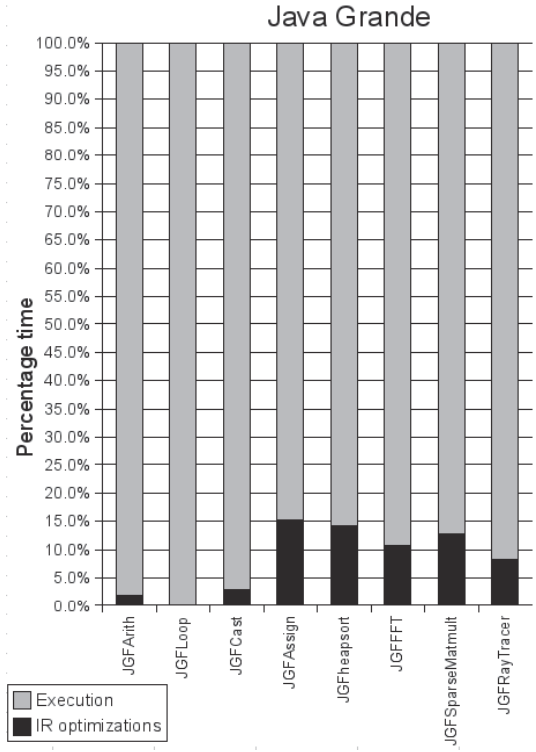

Figure 13: Time spent by ILDJIT over the compilation, optimization and execution phases using two cores; the translation phases are less than 0.4%

lation time for the Java Grande benchmark suite. It also show that for most of the time, ILDJIT executes the application code.

The significance of these results is that ILDJIT succeeds in parallelizing the compilation phases, as the translation and optimization phases overlap the execution time and are not an additive term of the total time.

5 Related Work

Two important related projects on parallel virtual machines should be mentioned.

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 [19] as well as in Intel's ORP [6] virtual machines there are parallel threads to perform code compilation and execution tasks.

We believe the structure ILDJIT to be rather distinct from such projects, and its use of parallelism and continuous optimization to be more aggressive.

6 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. Moreover since ILDJIT is a young system, we expect forthcoming releases will perform significantly better. ILDJIT is designed on a pipeline model for the translation and execution of CIL programs, where each stage (CIL to IR translation, optimization, IR to 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.

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. Another objective is to apply ILDJIT to multimedia application on embedded systems, including performance scaling and resource management. From a technical point of view, we are completing the implementation of the internal methods of the C# core libraries, the support for application threads and we are adding other optimization algorithms. Finally, currently ILDJIT targets the x86 architecture. In the future, support for other architectures, including ARM, co-processors and VLIW processors, will be added to allow better experimental evaluation on embedded multiprocessor systems.

Acknowledgement

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 Andrea Di Biagio for exception handling, and Ettore Speziale for his contribution to a garbage collector. Marco Cornero and Erven Rohou of ST Microelectronics have corroborated our commitment to free software development of dynamic compilation platforms.

References

[1] http://ildjit.sourceforge.net.

[2] Bea jrockit: Java for the enterprise technical white paper, 2006.

[3] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, 2002.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[6] 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.

[7] 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.

[8] ECMA, Rue du Rhone 114 CH-1204 Geneva. *Standard ECMA-335 Common Language Infrastructure (CLI)*, 3rd edition, June 2005.

[9] Nicu G. Fruja and Egon Borger. Analysis of the .net clr exception handling mechanism. In *Proceedings of the 2005 .NET Technologies Conference*, 2005.

[10] Java grande forum. http://www.javagrande.org/.

[11] Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

[12] 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.

[13] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 72–80, New York, NY, USA, 1999. ACM.

[14] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming.* New riders, 2001.

[15] B. Ramakrishna Rau. Levels of representation of programs and the architecture of universal host machines. In *MICRO 11: Proceedings of the 11th annual workshop on Microprogramming*, pages 67–79, Piscataway, NJ, USA, 1978. IEEE Press.

[16] Southern Storm Software. http://www.southern-storm.com.au. DotGNU Portable .NET project.

[17] Southern Storm Software. http://www.southern-storm.com.au/libjit.html. Libjit project.

[18] W. Richard Stevens. *UNIX Network Programming: Volume 2.* Prentice Hall, 1999.

[19] 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.

[20] Alfred V.Aho, Ravi Sethi, and Jeffrey D.Ullman. *Compilers Principles, Techniques and Tools.* Prentice Hall, 2003.

[21] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.