ILDJIT: A parallel, free software and highly flexible Dynamic Compiler

Simone Campanoni, Harvard University, 33 Oxford street, 02139 Cambridge, MA USA,
xan@eecs.harvard.edu
Michele Tartara, Politecnico di Milano, via Ponzio 34/5, 20133 Milano, Italy,
mtartara@elet.polimi.it
Stefano Crespi Reghizzi, Politecnico di Milano, via Ponzio 34/5, 20133 Milano, Italy,
crespi@elet.polimi.it

Abstract

ILDJIT, a new-generation dynamic compiler and virtual machine designed to support parallel compilation, is here introduced. Our dynamic compiler is a free software released through the GNU General Public License (version 2) and it 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. 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.
Keywords: Virtual machine, dynamic compilation, parallel system.

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, and then CLI (Common Language Infrastructure) are the de facto industrial standards for such virtualization of the Instruction Set Architecture (ISA). In particular CLI has become a very attractive framework, where applications written in multiple high-level languages, including also unmanaged language like C, 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 free software VES that we present here is named ILDJIT (Intermediate Language Distributed Just-In-Time) [5]. It is intended for multi-core 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. It takes as its input files written in the Common Intermediate Language (CIL) bytecode (designed by Microsoft for its DotNet framework and later standardized as ECMA-335 [8] and ISO/IEC 23271:2006 [10]) and executes them after translation to machine code by means of Just-In-Time compilation.
Parallelism is exploited in different ways: by means of a pipeline of translation and optimization phases (see Section 2), and by using a predictive approach, termed Dynamic Look Ahead (DLA, described in Section 3) to choose the methods to be compiled.

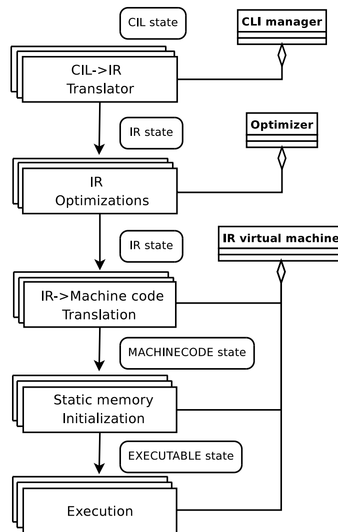

Figure 1: Pipeline model implemented inside the ILDJIT compiler

In particular, Section 2 and 3 presents ILDJIT's parallel features, Section 4 describes the recently obtained ILDJIT portability on different hardware architectures and the overall improvements it brought, and Section 6 presents some numeric results showing ILDJIT performances. Finally, Section 7 sets a road-map for future research and Section 8 concludes.

2 Pipeline architecture

The work of dynamic compilers can be divided into a series of subsequent steps, namely, loading the input bytecode, decoding it to an intermediate language, optimizing it, translating it to machine code and finally executing the program. Even if these steps expose a natural pipeline parallelism, most dynamic compilers, such as Mono and Portable.NET, perform these steps in a completely sequential way: each method of the program has to pass through all the phases of the compilation process, one after the other. When the method being executed needs to invoke another method, the execution will be stopped and the new method will have, in turn, to undergo the full compilation sequence.
ILDJIT exploits this kind of parallelism found across compilation phases by relying on an internal pipeline structure (see Figure 1) which aims to overlap as much as possible the compilation efforts. The software pipeline allows even sequential programs compiled in CIL to benefit from multiple hardware cores: while one core executes the current method, the other ones can be used to pre-compile and optimize methods that will have to be used in the future. Each method will have a flag indicating its current translation state, namely *CIL*, *IR*, *MACHINECODE*, *EXECUTABLE*. While the names of *CIL* and *IR* states are pretty self-explanatory, some word needs to be spent to clarify the meaning of the other ones: methods in *MACHINECODE* state are present in the system in CIL and IR form, and have already been translated to machine code. *EXECUTABLE* methods are present in the same forms of *MACHINECODE* ones, but all the static memory they need has been allocated and initialized, therefore they are ready to be executed.
Deciding which methods can be precompiled is done through DLA, explained in the next Section.

3 Dynamic Look Ahead compilation

Dynamic Look Ahead (DLA) [6] is a recent compilation technique, which aims to mask as much as possible the compilation delay by overlapping the compilation efforts. As Figure 2 shows, this technique predicts at runtime where the execution is going to go and it starts the compilation of parts of the application code before the execution asks for it.
This model is based upon the use of two priority queues (see Figure 3) (a low priority one and a high priority one) and on runtime code profiling of the bytecode application.

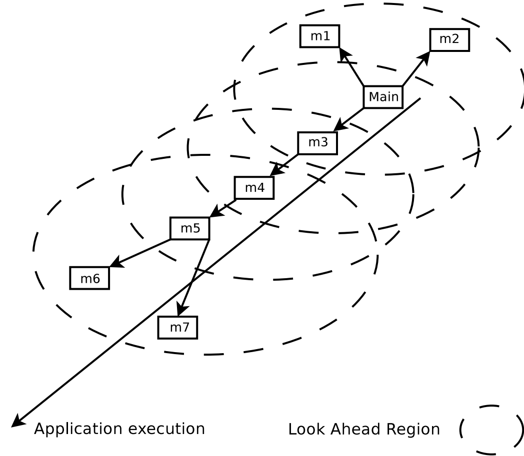

Figure 2: Look-Ahead-Region of DLA compilers

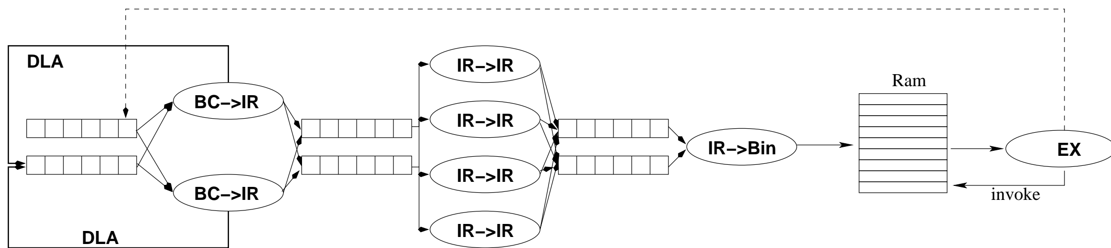

Figure 3: Internal organization of a DLA compiler. BC is the source bytecode (CIL) and ovals are the internal threads of the compiler

Most information comes from the Static Call Graph (SCG): it is the graph where each node represents a method of the program, and two nodes m_i and m_j are linked by a directed arc $m_i \to m_j$ if m_i can invoke m_j.
Even if the information of the SCG is static (therefore it exclusively depends on the source code of the program), the dynamic compiler does not know all of the SCG immediately: it gets to know it a portion at a time, while execution takes place. For this reason, the graph is defined Dynamically Known Static Call Graph (DKSCG).
Each time a method m is compiled, all the methods m_i it can invoke are candidates for being executed in the near future.
Let $\gamma(m, m_i)$ be the weighted distance between m and m_i. We define the Look Ahead Region as $LAR = \{m_i \mid \gamma(m, m_i) \leq Thr\}$, where Thr is an implementation-dependent threshold.
It is worth noting that the distance has to be weighted to take into consideration the probability of executing each method. The weight of the arc $a = (m_i, m_j)$ is defined as $f(\frac{1}{\lambda}, \delta)$ where f is a monotonic function of its parameters, λ is the likelihood of invocation of the method m_j from m_i, and δ is the *estimated time distance* between the execution of the first instructions of m_i and of m_j if the a arc is taken.
Methods in the LAR are added to the low priority queue to be precompiled, in an order depending on their execution probability. If during the execution of the program a not yet compiled method is invoked, a trampoline is taken, that calls an internal function of the compiler which will add the called method to the high priority queue (together with methods potentially invoked by it), in order to immediately compile it and resume the execution of the program.
As it can be seen, in an ideal situation, the high priority queue should never be used, and all the methods should already be ready when needed, thus completely masking out the delay introduced by JIT execution. However, it may happen that a wrong prediction leads to the need to insert a method in the high priority queue, or that compilation delay makes necessary to move a method from the low priority queue to the high priority one.

4 A multi-platform JIT

Another outstanding feature of ILDJIT is its portability: currently ILDJIT is able to run on top of ARM, x86 and AMD 64 bits processors. In this section we focus on the back-end developed for ARM processors.
In recent years, computational devices are changing. More and more frequently we find that smartphones, PDAs and other devices that are not usually associated with the concept of "computational power" are using increasingly powerful processors. In most cases, these processors are not compatible with Intel x86 Instruction Set Architecture, therefore, for a software to support them, it means that it has to be ported.
The development of ILDJIT is mainly done on Intel x86 processors, but the compiler has now been ported to ARM architectures too. This port has been done as part of the work for the Open Media Platform (OMP) European project, aiming at the definition of an "open and extensible service & software architecture for media-rich end-user devices, such as mobile phones or mobile media players, that will address software productivity and optimal service delivery challenges". [2]
The work led to the realization of a fully-working implementation able to run on ARM9 hardware with Vector Floating Point (VFP) coprocessor and, with partial functionality, even on hardware without VFP. At the moment, VFP is used just to perform floating point operations. Vector capabilities of this coprocessor are not yet used by ILDJIT.
ARM development is performed using the Qemu [1] emulator as the development platform, targeting an ARM926 processor. This choice is due to the ease of development offered by the emulated environment with respect to using a development board.
From time to time, results are validated by running ILDJIT on real hardware, namely an NHK15 development board by STMicroelectronics. This is needed because there is some difference between Qemu's emulated hardware and the real one, in particular with respect to the memory model. ARM9 hardware does not allow unaligned accesses to memory: each load or store operation has to be aligned to a multiple of the size of the data that are being read. This is due to architectural and performance reasons: in order to reduce the time to access memory, the power consumption and the cost of the processor, there is no hardware support for unaligned access. Failing to comply with this limitation, leads to having invalid data returned by load operations or saved by store operations. On the other hand, as of version 0.11, Qemu emulation permissively supports unaligned memory accesses. This is probably due the fact that Intel x86 processors support this kind of operation and Qemu directly uses the underlying primitives to interact with the emulated memory.
As of the end of the OMP project, ILDJIT is able to run on the NHK15 board JIT-compiling a Scalable Video Decoder translating an H.264 video to YUV format.
Porting ILDJIT to a second architecture brought several advantages: first of all, it increased the number of potential users and uses, and, due to the widespread adoption of ARM processors in the embedded systems industry, widened considerably the number of devices it can run upon.
Recently, multi-core ARM processors are starting to show up in commercial applications (such as the Cortex-A8 ARM dual core chip, used by Nokia's N900 Linux-based Internet tablet). ILDJIT's internal structure focused on the exploitation of hardware parallelism (already described in Sections 2 and 3) and based upon the use of many different operating system threads will automatically improve the performances of the compiler when run on such processors.

5 Project diffusion

The compiler ILDJIT is available on the web [4] since May 2007 and, after that date, more than one hundred releases have been provided. Table 1 reports the important improvements among the main releases of the project.
ILDJIT project started at Politecnico di Milano and currently both this University and Harvard University collaborate to this project.
Based on the number of downloads, it can be stated that after few months from the first release of the compiler, an increasing interest on the project has been shown. The number of downloads of the compiler increases month after month, and at the time this dissection has been written, more than ten thousands downloads have been registered. Moreover, Figure 4 shows the distribution of accesses to the website and downloads of the project over the world.
Several tutorials and documentations of internal aspects of the project are available within the several releases, in order to enable outside researchers and software developers to start to work within the project.

Table 1: Improvements of ILDJIT among its main releases

Release	Main improvements	Year
0.0.1	Start release	2005
0.0.1.2	Support of the 30% managed CIL provided	2006
0.0.1.7	Dead code and constant propagation optimizations provided	2007
0.0.1.8	5 code optimization algorithms provided	2007
0.0.1.9	2 Garbage collectors provided	2007
0.0.2	12 code optimization algorithms provided Support of the 40% managed CIL provided Support of the 10% unmanaged CIL provided	2008
0.0.3	Support of the 50% managed CIL provided Support of the 40% unmanaged CIL provided 16 code optimization algorithms provided 4 Garbage collectors provided Add the support of EM64T ISA Add the partial support of ARM ISA	2008
0.0.4	Add the partial support of Windows OS Add the partial support of MacOS OS Complete the support of ARM processors	2009
0.1.0	Support of the 70% managed CIL provided Support of the 80% unmanaged CIL provided 24 code optimization algorithms provided Add the partial support of Solaris OS	2009
0.2.0	Support of the 80% managed CIL provided Support of the 90% unmanaged CIL provided 24 code optimization algorithms provided Add the partial support of Solaris OS	2009
0.3.0	Support of the 100% managed CIL provided Support of the 90% unmanaged CIL provided 28 code optimization algorithms provided	2010

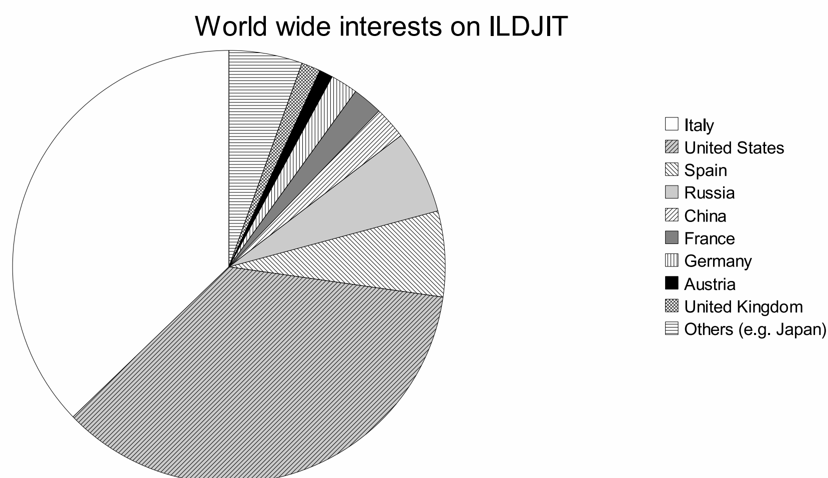

Figure 4: Snapshot of the world wide interests on ILDJIT project in April 2010

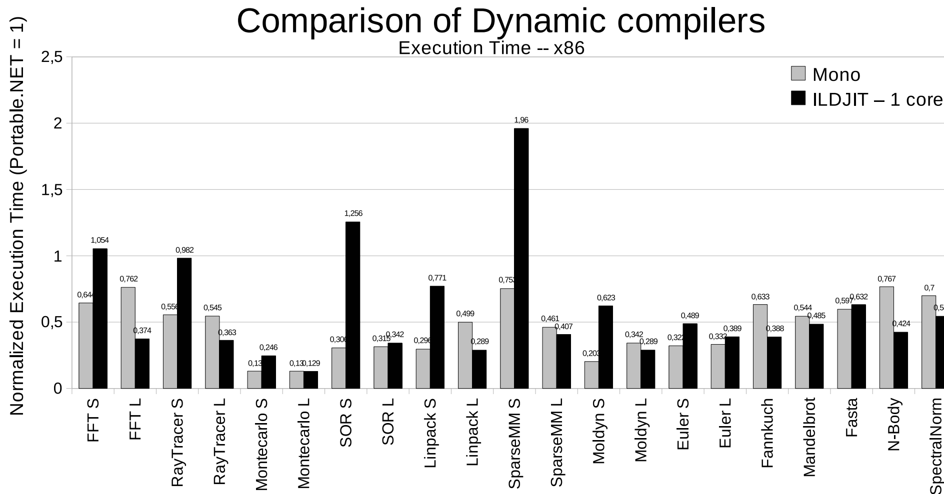

Figure 5: Execution time of Mono, Portable.NET and ILDJIT compiler. Benchmarks are considered with both small input set size (S) and large one (L).

Moreover, the ILDJIT project is currently used both at Politecnico di Milano and at Harvard University in teaching classes.

6 Experimental results

In this section, some measurement will be provided, comparing ILDJIT with the Mono and Portable.NET Just-In-Time compilers.
Intel x86 results were obtained by running on top of a Intel Xeon E5335 at 2 GHz. ARM results have been taken in Qemu 0.11, running on an underlying Intel Core 2 Duo P8400 dual core processor at 2.26GHz processor.
The numerical results reported are the mean value of four execution of the benchmarks out of five. The timings of the first execution have been discarded to prevent negative effects due to the caching of data to memory from the hard disk.

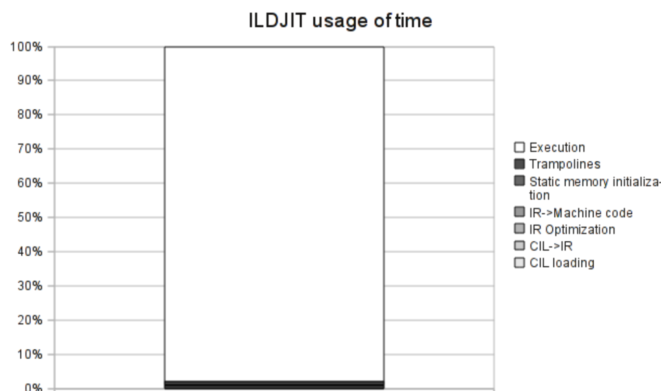

Figure 6: Distribution of times between the various phases of compilation for ILDJIT running Linpack

CIL loading	CIL→IR	IR Opt.	IR→EXE	Static mem. init.	Trampolines	Execution
0,27%	0,25%	0,61%	0,08%	0,15%	0,96%	97,67%

Table 2: Percentage of time required by each of the JIT compilation phases, for ILDJIT running Linpack (where “EXE” stands for “executable machine code“.

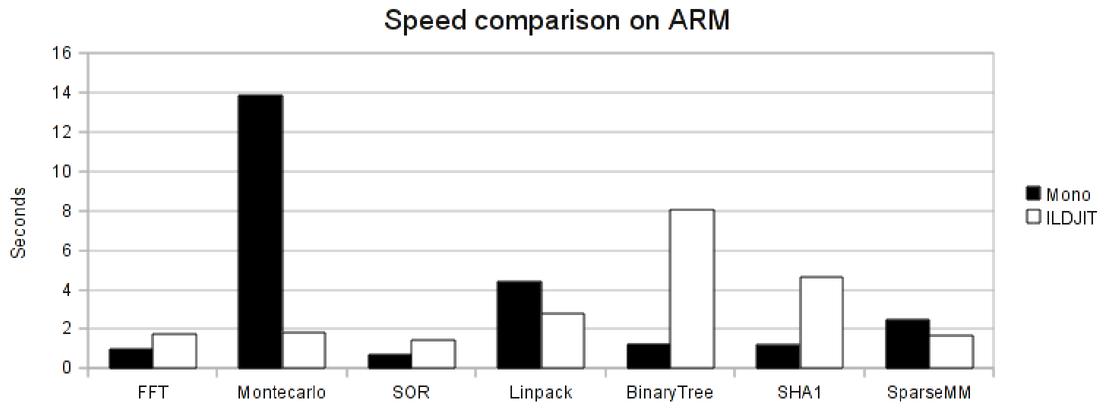

Figure 7: Comparison of ILDJIT and Mono running on ARM

	FFT	Montecarlo	SOR	Linpack	BinaryTree	SHA1	SparseMM
Mono	0,968	13,869	0,677	4,389	1,238	1,176	2,45
ILDJIT	1,709	1,842	1,442	2,784	8,072	4,622	1,659

Table 3: Execution times for some test programs with ILDJIT, compared with Mono on ARM platform

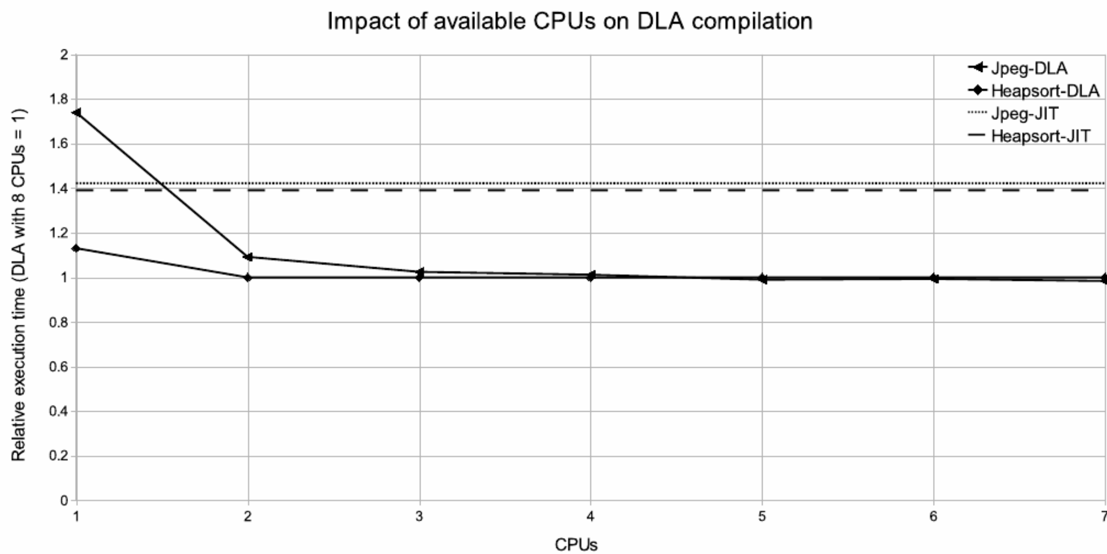

Figure 8: DLA results as a function of the number of CPU

The chosen benchmarks are taken from well-known test suites, such as Java Grande [9], Linpack and Scimark [11]. In particular, the following benchmarks have been used:

JGFFFT (FFT) part of the Java Grande benchmarks, it computes Fourier coefficients, which test the computation of floating point transcendental and trigonometric functions.

SciMarkMONTECARLO (Montecarlo) part of the SciMark test suite, it estimates π by approximating the area of a circle using the Montecarlo method.

SciMarkSOR (SOR) part of the SciMark test suite, it solves the Laplace equation in 2D by successive over-relaxation.

Linpack measures how fast a computer solves a dense $N \times N$ system of linear equations $Ax = b$. This test was originally written for Fortran. This particular version is in C# and comes from the *Pnet* [12] test suite.

BinaryTree taken from the *Great Computer Language Shootout* website [3], aimed at measuring the performance of various programming language, this test performs a series of operations on binary trees.

SHA1 it contains an implementation of the *Secure Hash Algorithm* (SHA), and in particular, of the SHA-1 variant, that produces a 160-bit message digest for a given data stream. Therefore, this algorithm can serve as a means of providing a "fingerprint" for a message. The test applies the SHA-1 algorithm on a series of test patterns.

JGFSparseMatMult (SparseMM) it is taken form the Java Grande Benchmark Suite and it exercises indirect addressing and non-regular memory references. An $N \times N$ sparse matrix (with a prescribed sparsity) is multiplied by a dense vector 200 times.

The first important evaluation is about the overhead of having a multi-threaded compiler: Figure 5 shows that even when ILDJIT runs on top of a platform that provides only one core, it is competitive w.r.t its competitors (Mono and Portable.NET). In order to compare the virtual machines, we have considered small inputs (benchmarks with the S character at the end of their name) and large inputs (L). We can notice that whenever the application code runs for less than 1 second (Small input), ILDJIT is slower than its competitors; this is due to the fact that the bootstrap of our compiler is more complex than others (it has to spawn threads, etc. . .) that are single thread program and therefore ILDJIT needs more time to start. On the other hand, if we consider application codes that run for more than 1 second (L), ILDJIT starts to be competitive.
As it can be seen in Figure 6 and in Table 2, overall JIT compilation times are just a tiny fraction of the total execution time.
Data in Table 3 and Figure 7 show a comparison between running times of some benchmarks with ILDJIT and Mono on the ARM platform. Mono [7] is the main open source JIT for CIL: it is a Virtual Execution System for programs in CIL bytecode, and it is developed by a wide community of programmers, with support from Novell.
As it can be seen in Figure 7, the results of the two compilers are quite close, with Mono frequently ahead. The outstanding results obtained by ILDJIT in some of the tests (such as the Montecarlo benchmark) come from the fact that its code generator is able to produce instructions for the VFP coprocessor, where available, therefore obtaining a great advantage in terms of performance on floating point based benchmarks. On the other hand, in other cases Mono is faster than ILDJIT, because of the better and more optimized code produced by its code generator when there are no floating point operation involved.
Figure 8 shows the effect of compiling a program with DLA with respect to JIT. The benchmarks used are Jpeg and Heapsort, taken from the MiBench suite. As it can be seen, DLA is strongly dependent on the number of used CPUs. With one CPU, it can be even slower than JIT compilation (because of the cost of handling the compilation pipeline) but as the number of CPUs increases, the result become increasingly positive. However, it has to be noted that the performance improvement is bounded. In particular, it depends on the branching factor of method calls in the compiled program. Once there are enough CPUs to precompile all of the methods, adding more CPUs does not bring any additional improvement.

7 Future work

At this time, ILDJIT focuses on exploiting parallelism internally, in order to make the JIT compilation as lightweight and as hidden as possible, through its software pipeline and DLA compilation. The executed program is not influenced by this.
In the future, some exploratory work will be done aiming at extracting parallelism from the executed code, in order to make its execution even faster and to allow legacy sequential code to take advantage of the increased degree of parallelism of the incoming architectures.

This could happen at two different levels, with detection of either fine-grained or coarse-grained parallelism (or even both). First of all, the code generator could be extended to support vector instructions, to perform Single Instruction Multiple Data calculations. On the other end, blocks of code that are not dependent on the rest of the program could be moved to a new thread and run on a different processor core, thus obtaining faster execution.

8 Conclusions

This paper presented the ILDJIT compiler and the features it has aimed at exploiting hardware parallelism present in recent architectures.
In particular, the software pipeline and DLA, built upon operating system threads, allow the compiler to adapt to the underlying hardware, fully exploiting its resources, even when running programs non-explicitly written for parallel systems.

References

[1] Qemu - open source processor emulator. http://www.qemu.org.

[2] Open Media Platform (OMP) European project. http://www.openmediaplatform.eu, 2008.

[3] The Great Computing Language Shootout. http://shootout.alioth.debian.org/, 2009.

[4] S. Campanoni. ILDJIT website, December 2009.

[5] S. Campanoni, G. Agosta, S. Crespi Reghizzi, and A. Di Biagio. A highly flexible, parallel virtual machine: design and experience of ildjit. *Softw. Pract. Exper.*, 40(2):177–207, 2010.

[6] S. Campanoni, M. Sykora, G. Agosta, and S. C. Reghizzi. Dynamic look ahead compilation: A technique to hide jit compilation latencies in multicore environment. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction*, pages 220–235. Springer, 2009.

[7] M. de Icaza, P. Molaro, and D. Maurer. http://www.go-mono.com/docs. Mono documentation.

[8] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fourth edition, June 2006.

[9] Edinburgh Parallel Computing Centre. Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/research/javagrande/benchmarking.html.

[10] ISO-IEC. *Programming Languages—C#, ISO/IEC 23270:2006(E) International Standard*, ansi standards for information technology edition, 2006.

[11] R. Pozo and B. Miller. http://math.nist.gov/scimark2. SciMark benchmark.

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