Parallelism and Retargetability in the ILDJIT Dynamic Compiler

Michele Tartara, Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi
Politecnico di Milano, Italy

Abstract

Modern computer architectures are becoming increasingly parallel with each generation. At the same time, new, different and binary incompatible platforms are becoming more and more widespread on the market. This paper presents ILDJIT, a Just-In-Time compiler that aims at exploiting parallelism even when dealing with non-explicitly parallel programs, and the advantages obtained by introducing portability.

1 Introduction

Users of modern computing systems have come to expect faster and faster hardware, at each new generation, and so do software developers, in need of constantly increasing computational power.
Hardware manufacturers have been able to comply with this request thanks to the fact that Moore's Law, predicting a doubling of the number of transistor on an integrated circuit approximately every two years, has proven to be quite an accurate model of the advancements of chip manufacturing technology.
Unfortunately, bigger challenges are being faced every day, while physical limits of the materials draw near. The constant increasing of processors speed, measured in MHz, has come to an end. Every advancement must now come from architectural improvements, that are not easy to obtain.
In this scenario, until now and for the foreseeable future, the easier and most effective way to improve performances has proven to be increasing the number of hardware cores, thus incrementing the degree of parallelism of processors. Unfortunately, this kind of improvement does not automatically increase the performances of existing software, or of new programs written in a sequential fashion.
In order to fully exploit these new multi-core architecture and the future many-core ones, new programs have to be written according to a parallel paradigm, and algorithms have to be thought with parallelism in mind.
Traditional Just-In-Time (JIT) compilers executing single threaded applications usually exploit only one processor core. ILDJIT (Intermediate Language Distributed Just-In-Time) is the research compiler designed at Politecnico di Milano, focused on deep exploitation of parallelism in multi-core machines.
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 [6] and ISO/IEC 23271:2006 [8]) 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.
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 5 presents some numeric results showing ILDJIT performances. Finally, Section 6 sets a road-map for future research and Section 7 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.
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.
In ILDJIT, this does not happen, because loading CIL bytecode, translating it to ILDJIT's Intermediate Representation (IR), applying optimization algorithms upon the IR, translating IR code to machine code and, finally, executing the program are parallel operations, executed in different operating system threads (POSIX threads). Each of the threads implements one of the compilation steps, and each compilation step can be implemented by more than one thread (in order let it be performed on different methods at once).
The various threads operate as a software pipeline: each of them performs one step of the compilation process on a different method of the program.
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

Dynamic Look Ahead (DLA) [4] is the the abstract model that describes the way ILDJIT uses to choose the methods that have to be precompiled.
DLA compilation is effective when the number of available processors is at least equal to the number of threads executing, translating or optimizing code.
This model is based upon the use of two priority queues (a low priority one and a high priority one) and on statically and dynamically available information about the program.
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 $m_i \rightarrow m_j$ 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.
Pushing the concept of DLA to the limit leads to Ahead of Time compilation, where all of the methods are precompiled, obtaining performances comparable to those of statically compiled programs.

4 A multi-platform JIT

Another outstanding feature of ILDJIT is its portability.
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.

More than this, the quality of the source code was greatly improved during the porting work: some fragments of code, initially platform specific, have been rewritten to be more adherent to ANSI C, with less platform-specific assumptions. This brings as a bonus the possibility of introducing support for further platforms in the future, with a reduced implementation effort.

Also, having to run the compiler on a more constrained architecture such as ARM, pushed us to greatly improve the code to better exploit the available resources, reducing its memory footprint by up to two thirds and its execution time by up to nearly one half, as shown in the next Section. Most of this code rewriting has taken place in the platform-independent part of ILDJIT, thus affecting performance both on ARM and on x86 processors. This sensible improvement is mainly due to changes in the data structures used to represent the programs being compiled by ILDJIT while in IR form. Removing many function pointers from those structures, and substituting them with a better use of `include` preprocessor directives, led to a great reduction of the occupied memory space (since each structure is used many times, thus magnifying the effect of each modification). Moreover, less function pointers means less indirections through memory when performing function calls, and therefore more speed. Other speed improvements came from posticipating the initialization of some fields to the moment when it was strictly required: this way, if a field is not used, it is just cleared setting it to a `NULL` value, but no time is wasted initializing it. Despite the work done, the amount of required memory is still greater than that of Mono, but there is space for further improvement, to make ILDJIT even more apt to embedded systems.

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 Experimental results

In this Section, some measurement will be provided, showing the performance gain obtained after completing the porting to the ARM architecture and the cleanup of the code base that was required by the reduced resources available on these systems.

x86 results were obtained by running on an Intel Core 2 Duo P8400 dual core processor at 2.26GHz. ARM results have been taken in Qemu 0.11, running on the same underlying Intel hardware.

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.

	FFT	**Montecarlo**	**SOR**	**Linpack**	**BinaryTree**	**SHA1**	**SparseMM**
ILDJIT	0,048	0,072	0,042	0,081	0,202	0,18	0,054
Old ILDJIT	0,075	0,098	0,067	0,103	0,259	0,304	0,08
Speedup	36,00%	26,53%	37,31%	21,36%	22,01%	40,79%	32,50%

Table 1: Execution times for some test programs with the old version of ILDJIT and the new one

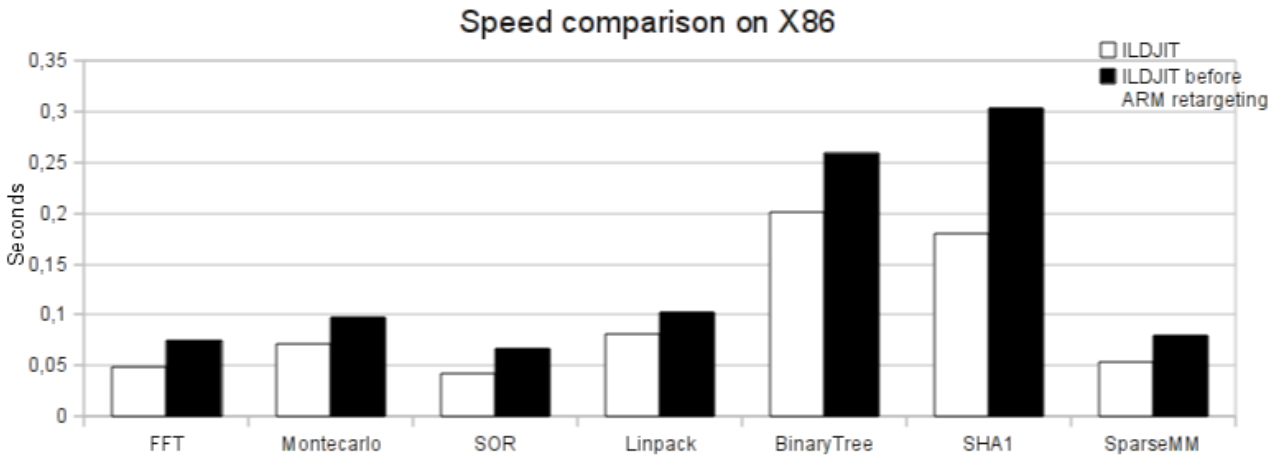

Figure 1: Performance improvement of ILDJIT running on x86

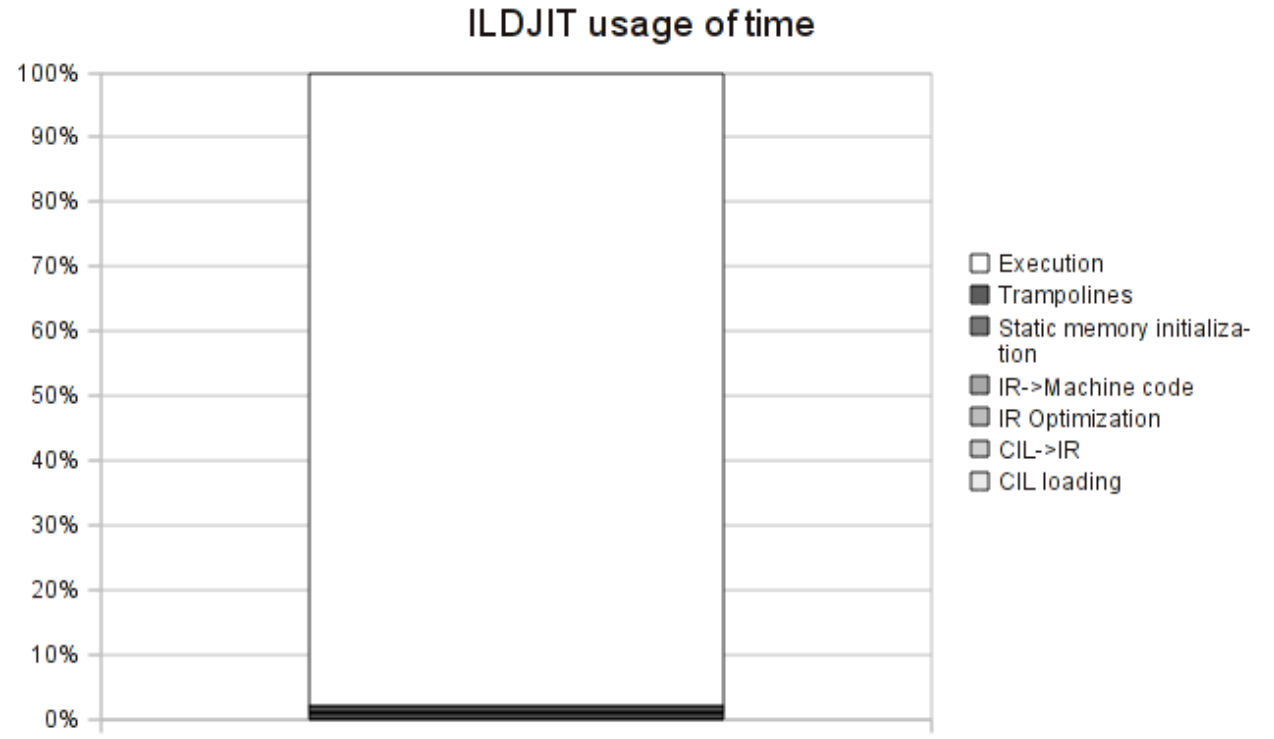

Figure 2: Distribution of times between the various phases of compilation for ILDJIT running Linpack

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".

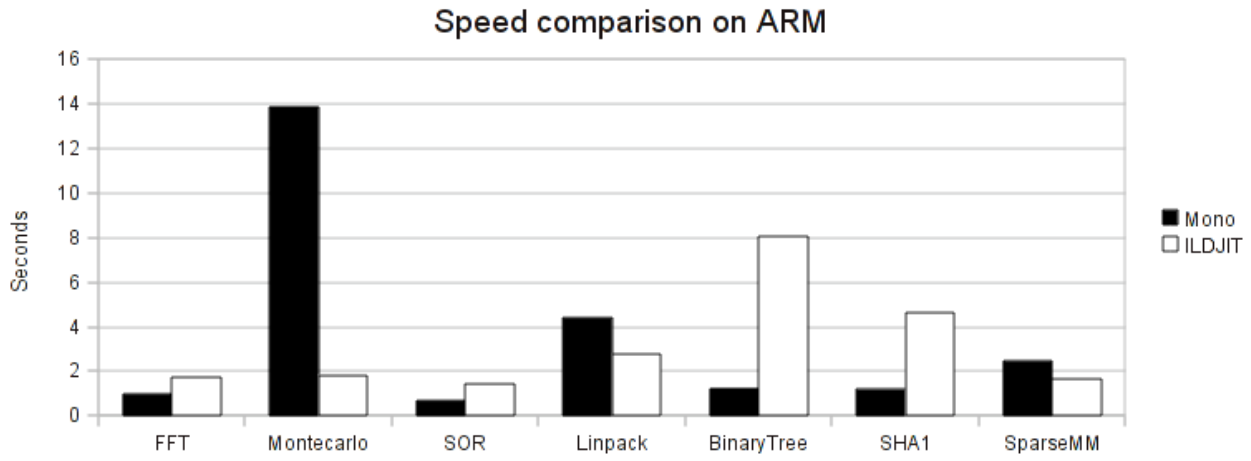

Figure 3: Comparison of ILDJIT and Mono running on ARM

	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

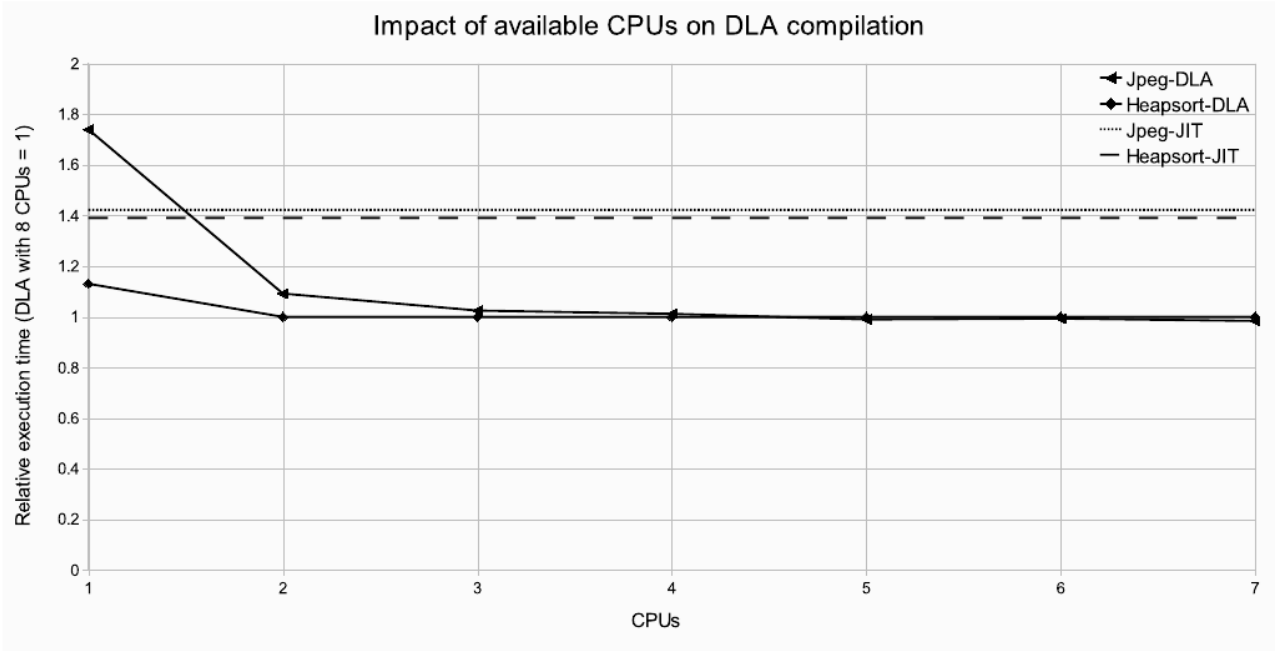

Figure 4: DLA results as a function of the number of CPU

The chosen benchmarks are taken from well-known test suites, such as Java Grande [7], Linpack and Scimark [9]. 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 Sci-

Mark 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* [10] 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.

As it can be seen in Table 1 and Figure 1, the modifications led to greatly improved performances.
Looking at the bar diagram, it is clear that the speed gain is constant, and does not depend on the specific program being compiled.
As it can be seen in Figure 2 and in Table 2, overall JIT compilation times are just a tiny fraction of the total execution time.
Data in Table 3 and Figure 3 show a comparison between running times of some benchmarks with ILDJIT and Mono on the ARM platform. Mono [5] 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 3, 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 4 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.

6 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.

7 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.
Moreover, ILDJIT's portability has been described, showing the benefit that introducing support to a second architecture brought to the compiler in general.

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, 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.

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

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

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

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

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

[10] Southern Storm Software. http://www.southernstorm.com.au. DotGNU Portable .NET project.