ILDJIT: a parallel dynamic compiler

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[1], is able to achieve significant speedups (1.183 on average) with respect to the baseline, when the underlying hardware exposes at least two cores.

I. Introduction

Dynamic translators and optimizers (DTOs) [2] are nowadays the established way to provide object code compatibility between different architectural platform, as well as interoperability between different high level languages.

Most works in the field have targeted traditional single-processor and single-core architectures. However, multiprocessor architectures are becoming the most viable way to improve performance, since increasing the clock frequency leads to unacceptable power consumption and manufacturing cost growths [4]. Thus, this paper focuses on obtaining higher performances in DTOs on parallel 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.* [9]:

> *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 itself 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 should be implemented as external modules loadable at runtime (plugins).

In this paper, we provide two major contributions: the definition and experimental evaluation of a parallel compilation and optimization model. The second one is the definition of an execution model for a DTO 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 [7]. The current name of the project is ILDJIT. One could fear that a compiler implemented as a distributed and dynamically linkable program would have to pay a high overhead, to the point that the benefit from hardware parallelism might be offset especially for a small number of processors. On the contrary our experiments show that the performance of applications compiled by our DTO are comparable to some of the best known dynamic compilers, on a single processor machine, and superior on multi processor platforms.

The paper is organized as follows. In Section II we outline the execution model of ILDJIT dynamic compiler. In Section III 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.

II. 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.

A. Translation unit and Intermediate representation

Choosing the correct granularity for the translation process is especially important in a dynamic compiler [6]. We have chosen the method as translation unit. ILDJIT can 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

[1]This work is supported in part by the European Commission under Framework Programme 7, OpenMedia Platform project

[2]We refer to the classification and terminology for dynamic compilers proposed by Rau [11] and Duesterwald [6]

code across different PUs which may be connected through slow communication channels.

To minimize the communication costs, we have designed our own intermediate representation (IR), with the following goals: first, to provide a compact representation of individual methods (CIL offers a compact representation of the whole program, but spreads information related to each method across the assembly); second, to offer a machine-neutral, register-based language, such that each construct has a clear and simple meaning. IR instructions are composed by pseudo assembly instructions (e.g. IRADD, IRSUB ...) and by pseudo object oriented bytecode (e.g. IRNEWOBJ). By this way many optimization algorithms that aim to compute static properties of the program are helped (e.g. memory alias analyzers).

Translation is therefore split in two phases: first the CIL fragment is translated into IR, then IR is translated into the target machine code. 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. ILDJIT resolves the linking problem by a lazy compilation [8] implementing the *trampolines*.

B. 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.

The pipeline model exposes five stages as shown in Figure 1. 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.

The pipeline model is organized into the four thread groups depicted in Figure 2. Each group implements one of the first four stages of the pipeline (Figure 1).

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 initialization), 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 3. 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 causes a deadlock is the

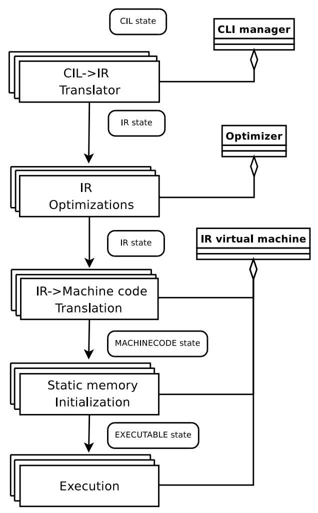

Fig. 1. The translation pipeline model: on the left, the translation pipeline stages; on the right, the modules that implement the various stages; in the rounded boxes the state of the method code at the beginning of each stage. Each method can be in the following non-overlapping *translation states*: 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.

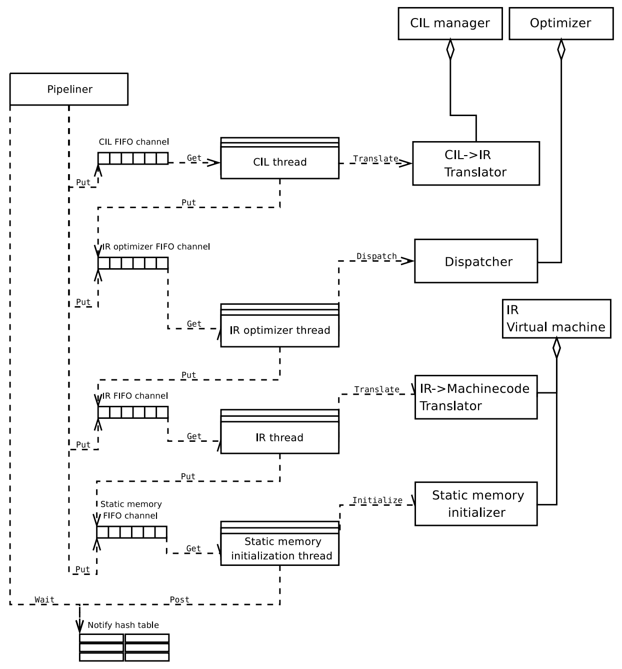

Fig. 2. The Pipeliner's structure: in the middle four groups of boxes represent four groups of threads, each one assigned to a compilation task; on the right, the modules that implement the various compilation tasks.

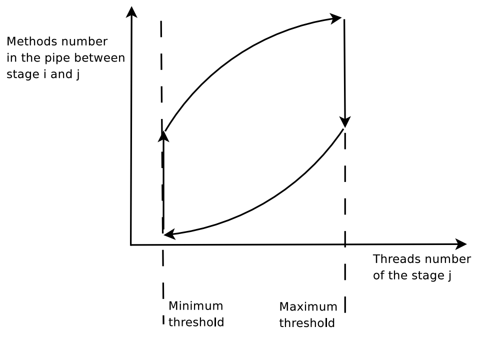
Methods number in the pipe between stage i and j
Threads number of the stage j
Minimum threshold
Maximum threshold

Fig. 3. Histeretic pattern used for adaptation of thread numbers.

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 $S+1$ 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 (it means thread i waits the end of the compilation/execution of method M_{i+1});

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.

C. Optimizations

ILDJIT allows optimizations both at IR and target code level. The rationale 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 [2].

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.

III. Experimental evaluation

To compare the baseline ILDJIT system against its main competitors, Mono and Portable.NET, we have chosen the C# version of the well-known benchmark suite *Java Grande Forum Benchmark Suite* [10]. Figures 4 report the execution times of the selected benchmarks running on the three VES

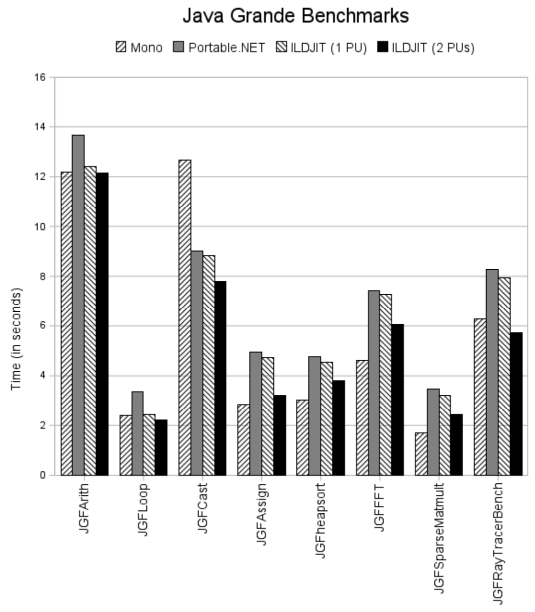

Fig. 4. Total execution time of the JGrande benchmarks run by Mono, Portable.NET and ILDJIT; the speedup of ILDJIT in the 2 PUs hardware configuration is due to the compilation phase overlapping done by the VM

implementations. ILDJIT results are provided for both a single PU and two PUs machine – the other VMs do not get any advantage by running on a two PUs machine.

Since ILDJIT uses the same code generation and core libraries as Portable.NET, it can be seen that the overhead of its multi-threaded architecture is limited, as ILDJIT always outperforms Portable.NET. On the other hand, Mono is faster due to the limitations of the code generator – the native code produced by ILDJIT is very similar to that produced by Portable.NET (though slightly smaller due to high-level optimization), and both only produce a simple code not tuned to the specific Intel instruction set, while Mono can rely on a much more advanced code generator. However, overlapping computation and compilation in the two PUs setting allows ILDJIT to achieve an average speedup of 1.183, which brings it performance on par with those of Mono.

Here we present a breakdown of the execution time of the Java Grande benchmarks among the various phases of the compilation and execution. As Figure 5 shows, on a single processor machine ILDJIT already spends most of the time executing target machine code, but a significant part of the time is spent optimising the IR methods. It is notable that the translation between IR and machine code is extremely fast (less than 0.1% of the execution time) – thus, we can say that the intermediate representation does a good job in allowing most of the translation to be performed at a machine independent level. On the other hand, Figure 6 shows that two PUs are sufficient to overlap the translation time for the Java Grande benchmark suite. Basically, the user does not perceive any compilation overhead.

IV. Related Work

Due to the considerable impact of the Microsoft ".NET" initiative, several projects aimed at VES implementation.

Microsoft itself released three implementations of VES: the .NET framework (for desktop environments), the Compact

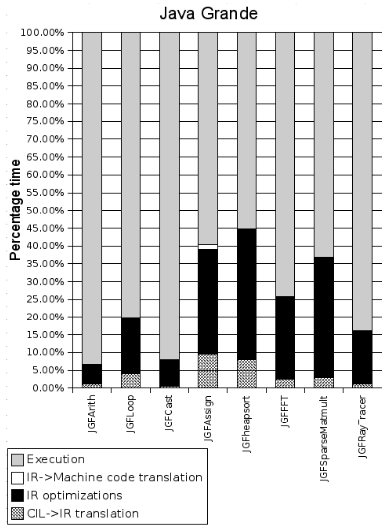

Fig. 5. Time spent by ILDJIT over the compilation, optimization and execution phases using a single PU; the translation from IR to machine code is less than 0.1%

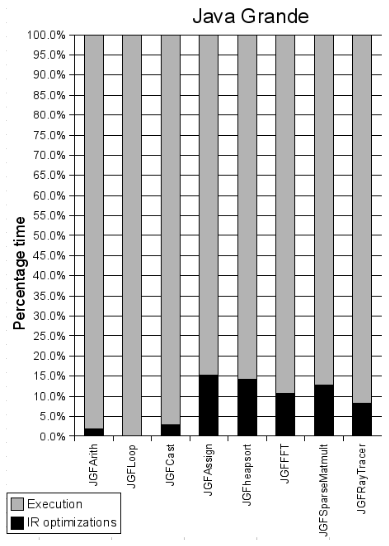

Fig. 6. Time spent by ILDJIT over the compilation, optimization and execution phases using two PUs; the translation phases are less than 0.4%

Framework (for embedded devices) and Rotor, a "shared-source" implementation. Mono is an open-source project led by Novell (formerly by Ximian) [5] to create an ECMA standard compliant .NET compatible set of tools, including a C# compiler and a Common Language Runtime. Portable.NET is an free software implementation of CLI by Southern Storm [12]. Its primary design goal is portability to as many platforms as possible; such goal is achieved through use of interpretation rather than Just-In-Time compilation as default execution method.

In BEA's JRockit [1] virtual machine, methods are compiled without performing code optimizations for their first execution; the compilation is 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 [13] as well as in Intel's ORP [3] 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.

V. 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/IR translation, optimization, IR/native translation, and execution) can be performed on a different processor. This choice brings a great potential for continuous and phase-aware optimization in the domain of server applications, as well as fast reaction times and effective compilation on embedded multiprocessor systems.

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.

References

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

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

[3] 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.

[4] Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea C. Ornstein, and Erven Rohou. An experimental environment validating the suitability of cli as an effective deployment format for embedded systems. In Per Stenström, Michel Dubois, Manolis Katevenis, Rajiv Gupta, and Theo Ungerer, editors, *HiPEAC*, volume 4917 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2008.

[5] Miguel de Icaza, Paolo Molaro, and Dietmar Maurer. http://www.go-mono.com/docs. Mono documentation.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.