Dynamic Look Ahead Compilation: a technique to hide JIT compilation latencies in multicore environment *

Simone Campanoni**, Martino Sykora, Giovanni Agosta and Stefano Crespi Reghizzi

Politecnico di Milano, Milano 20133, Italy,
{campanoni,sykora,agosta,crespi}@elet.polimi.it
http://compilergroup.elet.polimi.it

Abstract. Object-code virtualization, commonly used to achieve software portability, relies on a virtual execution environment, typically comprising an interpreter used for initial execution of methods, and a JIT for native code generation. The availability of multiple processors on current architectures makes it attractive to perform dynamic compilation in parallel with application execution. The major issue is to decide at runtime which methods to compile ahead of execution, and how much time to invest in their optimization. This research introduces an abstract model, termed Dynamic Look Ahead (DLA) compilation, which represents the available information on method calls and computational weight as a weighted graph. The graph dynamically evolves as computation proceeds. The model is then instantiated by specifying criteria for adaptively choosing the method compilation order. The DLA approach has been applied within our dynamic compiler for .NET. Experimental results are reported and analyzed, for both synthetic programs and benchmarks. The main finding is that a careful choice of method-selection criteria, based on light-weight program analysis and execution tracing, is essential to mask compilation times and to achieve higher overall performances. On multi-processors, the DLA approach is expected to challenge the traditional virtualization environments based on bytecode interpretation and JITing, thus bridging the gap between ahead-of-time and just-in-time translation.

1 Introduction

Portable, byte-code based, Object Oriented languages such as Java, Python and C# have achieved widespread adoption in both industry and academia. Modern Virtual Machines (VM) frequently include a dynamic translation system, the *Just In Time* (JIT) compiler. A JIT compiler translates a byte-code portion (typically a method) to native binary code, when needed. The generated binary code is then executed every time it is required. Dynamically compiled code can achieve large speedups, especially in the long run, since the execution time of a native method is dramatically lower than that of an interpreted one. However, the performance of a JIT-based VM is still lower than that of native code produced by static byte-code compilation [23], or *Ahead Of Time* (AOT) compilation. The loss of performance is due to compilation overhead – often called *startup time* – and to the poor quality of the generated code, since the startup time minimization prevents the aggressive and costly optimizations usually performed by static compilers.

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

** This author is supported in part by the ST Microelectronics

At the same time, multi-core technology is being employed in most recent high-performance architectures as a way to provide more computational power without relying on the reduction of the clock cycle, which is becoming increasingly difficult due to technology limitations.

Thus, we consider a multiprocessor environment and study how specialized threads of a dynamic compiler can compile bytecode portions in advance, in parallel with the application execution. In a best case scenario, there is no compilation overhead, because compilation fully overlaps with execution and methods are already compiled when they are invoked. Moreover, optimizations are applied to provide high quality code. Our goal is to prove that, given enough hardware resources, it is possible to effectively mask the compilation delays, approximating the ideal case shown in Figure 1, where compilation threads `Th1` and `Th2` – running on processors P_1 and P_2 – supply the requested native methods to the execution thread in advance. Compilation times depend both on method size and on the optimizations applied. To reach this ideal case, the dynamic compiler should predict the execution trace, and be able to recognize hot-spots. We call such a compiler a *Dynamic Look Ahead Compiler*.

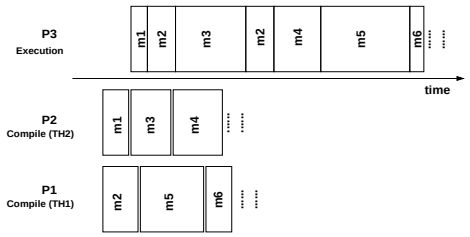

Fig. 1: An ideal case. Each invoked method has already been compiled and optimized.

While a processor is executing a method, compilation threads (running on different processors) *look ahead* into the call graph, detecting methods that have good chances to be executed in the next future. Moreover, they guess whether a method is an *hot spot* or not, and apply aggressive optimizations accordingly. Hence, DLA compilation dynamically exploits static code properties (call graph, structure of the method) for execution trace prediction and hot-spot optimization.

The DLA compilation paradigm, conceived for multiprocessor architectures and object-oriented languages, is the main contribution of this paper. In the rest of the paper, we outline the theoretical model in Section 2 and describe DLA compilation in Section 3. Section 4 reports the experimental results. Section 5 provides a survey of prior works, highlighting the distinctive aspects of the DLA compilation. Conclusions are discussed in the last Section.

2 Model

A DLA compiler examines the methods to be compiled with the aim of deciding:

Compilation order In which order methods should be compiled. The *compilation order* quality is measured by its similarity to the actual execution order of the methods – considering only the first call of each method, since no compilation is required for further invocations.

Optimization level Which optimizations should be applied in compiling. To this end, a *level of optimization* is assigned to each method.

Different platforms may use different criteria for dispatching and fine-tuning methods compilation. For a program, the basic concept is the *Static Call Graph* $SCG = (M, I)$, where M is the set of *methods* and I is the set of possible *invocations*. A direct arc $a = (m_i, m_j) \in I$ connects method m_i to m_j if the former may call the latter at run-time. The `main` method belongs to M and is named the *root*. The set of immediate successors of a method $m \in M$ is $S(m)$.

Initially, the SCG is not known to the DLA compiler, which progressively discovers it. We call this graph the *Dynamically Known Static Call Graph (DKSCG)*. Thus, the DKSCG is the portion of the SCG that is dynamically known: each time a method m is compiled, the DKSCG is updated with the subset of $S(m)$ not yet compiled.

Next we enrich the DKSCG with arcs and node weights, summarizing the relevant properties for deciding compilation order and optimization level. Figure 2 shows a portion of a generic DKSCG, where w and w' are the weights assigned to the arcs and nodes, respectively.

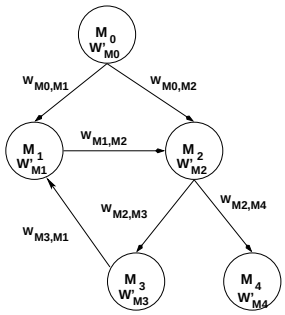

Fig. 2: Dynamically Known Static Call Graph

First consider the weight-less graph. Knowledge of the set of successors of a method gives some hints on the compilation order. It is obvious that, when a method is running, its immediate successors are likely to be executed soon. Following this assumption, the methods can be ordered according to their distance from the root. Let m be a method involved in the compilation process and n be the methods ordered so far; not yet compiled methods in $S(m)$ will be ordered starting from $n + 1$, as their appear in m body.

However such an ordering is rather unsatisfactory, as it neglects the effect of conditional branches – the execution order of the successors of a method depends on the control flow and input data. Adding information on the likelihood of the execution of each method, can improve the ordering quality. To this end, the model is enriched with weights. An arc $a = (m_i, m_j)$ of the DKSCG is characterized by two attributes: the

likelihood of invocation λ, i.e. the likelihood that a is taken after execution reaches node m_i; and the *estimated time distance* δ between the execution of the first instructions of m_i and of m_j, if that arc is taken.

The weight of an arc $a = (m_i, m_j)$ is defined as $w_a = f(\frac{1}{\lambda}, \delta)$ where f is a monotonic function of its parameters. Hence, given a method m, the not yet compiled methods in $S(m)$ can be ordered by increasing arc weights.

For a node m_i, let $\gamma(m, m_i)$ be the weighted distance from the executing method m. We here define the so called *Look Ahead Region* (LAR) as $LAR = \{m_i | \gamma(m, m_i) \leq Thr\}$, where Thr is an implementation dependent threshold. LAR should contain those methods having good chance to be executed in the next future. A method is a candidate for compilation if it belongs to the LAR. In this case it is enqueued for compilation, with an order depending on f. The weights on arcs dynamically change their values, as well as LAR. Details about LAR updating are provided in Section 3.

The weights on arcs must be combined with the information on the computational load of methods, providing hints on the most appropriate level of optimization. To achieve this, to each method m is given attribute indicating the computational load, t_{exc}. The weight of the node is a monotonic function of the attributes: $w'_m = f'(t_{exc})$ By convention, the higher w'_m, the higher is the benefit due to an aggressive optimization of m.

Note that the proposed general model may have different implementations, depending on: the definition of functions f and f'; the way the function arguments are computed. In the sequel, we present two model implementations, integrated in our DLA compilation framework [7]. A *naive* one, where λ, the likelihood of invocation, is dropped; δ is the order of appearance of a method into the bytecode and $f(\frac{1}{\lambda}, \delta) = \delta$. A more refined implementation, where static branch prediction techniques [5] are used to estimate the parameters λ, δ and the function f. For both models f' depends on hot-spot detection and is defined as $f'(t_{exc}) = t_{exc}$. Our implementation closely follows [5]. On the set of benchmarks used in Section 4 the branch predictor achieves a missrate of 18% comparable to the 20% declared in [5].

3 Dynamic Look Ahead Compilation

In this section, we focus on the DLA principle, presenting the application scenario and analyzing the main problems: execution trace prediction and hot spot detection. Specific choices concerning the definition of the main components of the model – functions and parameters – are also discussed.

Figure 3 shows the control flow of a DLA system, composed of several threads (shown as ovals) connected by queues and composing a compilation pipeline. First the methods are pushed into a compilation queue and translated from bytecode (BC) to an intermediate representation (IR). Then multiple threads, running onto multiple processors, optimize the IR methods and provide them for a final step of translation toward native code. Native methods, when ready, are installed in memory and invoked when needed. A method can be pushed for compilation in two cases: it is required for the execution but it has never been compiled (dashed arc in figure); it is detected by the DLA system as a method with high chances to be executed soon (bold arcs). The DLA decision is taken in the first stage, where the DKSCG is updated with new weighted nodes and the pipeline is supplied with new methods.

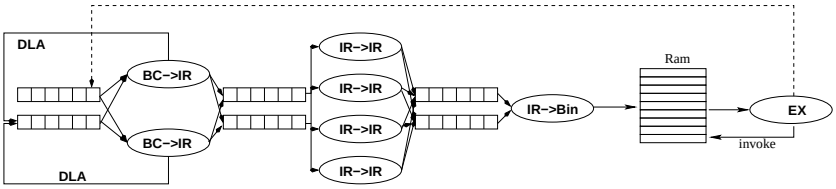

Fig. 3: DLA in a Pipelined Compilation Framework: the framework shown takes as input bytecode (BC) produced from source files, and uses an intermediate representation (*IR*) to perform machine independent optimization. The pipeline is based on a priority queue implemented by pairs of FIFO queues. Priorities of individual methods can change on information discovered at runtime. The execution goes through the trampoline if the called method has not been compiled yet.

Two queues with different priority are shown in Figure 3. The low priority queue contains those methods detected by the DLA engine as the most likely candidates for execution in the near future. The high priority one contains the method that is presently required for execution and the methods potentially invoked by it. Ideally, the high priority queue should always be empty, since all the invoked methods should be provided as native code in advance. However, the prioritization mechanism is useful when – due to wrong prediction or compilation delay – an invoked method has not yet been compiled (thus it has to be enqueued with high priority or moved from low to the high priority queue – *method prioritization*).

3.1 Applicative Scenario and Technique

DLA compilation is effective when the number of available processors is at least equal to the number of threads dedicated to execution, compilation and optimization, to avoid threads switching overhead. In this paper, for the sake of clarity, we focus on single thread applications. Thus, only one processor is dedicated to the execution and the remaining ones are exploited for compilation and optimization.

Let us consider the first invocation of a method in a typical JIT execution. The control flow jumps to a code fragment known as *trampoline*, which yields control to the dynamic compiler. The dynamic compiler, in turn, generates (and possibly optimizes) the native binary code, then replaces the trampoline with the address of the generated binary. In the DLA compilation the dynamic compiler also prepares other methods for parallel compilation. To this end, the compilation routine *looks ahead* into the portion of the SCG seen by the method it is currently processing, i.e. composed of its children methods. They are added to the DKSCG and, if they belong to the LAR, they are pushed into the *compilation queue*, in an order depending on the underlying model. Conceptually, it is equivalent to an assignment of weights to the DKSCG arcs, in accordance with the function presented in Section 2. The queue elements are consumed by one or more compilation and optimization threads, running in parallel with the execution flow and distributed over multiple processors. Each dequeued method is compiled and optimized, making it ready for the execution as soon as possible. During its compilation, the above process is iterated.

If the DLA compilation is well tuned, the LAR is constantly updated, with the aim of (i) compiling methods in advance; (ii) controlling the pressure on the compilation queue. Figure 4 shows a DLA compilation thread in the large.

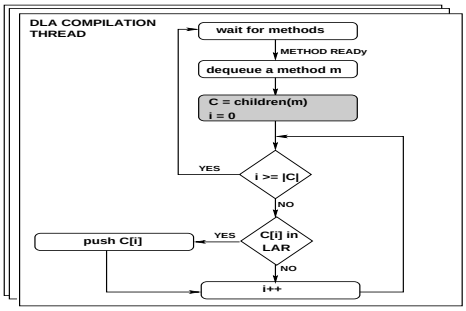

Fig. 4: DLA compilation thread(s), shown as multiple boxes. Each of them wait for methods in the compilation queue. When a method `m` is ready, it is dequeued and compiled. The set C of its children is then computed, as shown by the third stage (shaded in light grey), where the *look ahead* process is effectively performed. Elements of C belonging to the LAR and not yet compiled are pushed into the compilation queue. The update of the DKSCG is not explicitly shown in figure, as well as the pushing order is not highlighted.

Summarizing, the DLA compilation tries to compile in advance (exploiting hardware parallelism) those methods that will be useful in the near future. To make predictions on the execution flow, compilation threads: (i) build and update the DKSCG; (ii) keep information about the Dynamic Call Graph (DCG), the SCG subgraph of the methods effectively executed; (iii) keep information about the execution trace, which is a linearization of the DCG. Both the execution trace and the DCG need a tracing mechanism (e.g. trampolines). In absence of this mechanism we observe a loss of information. (iv) update the LAR, which both limits the pressure on the compilation queue and drives the prediction. Figure 5 shows the relations between these concepts. Since a

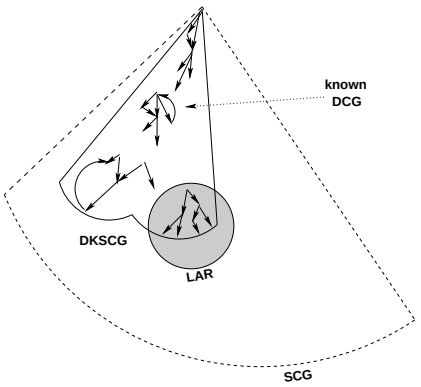

Fig. 5: Information exploited in DLA compilation. The SCG, unknown at run time, is the region bounded by dotted lines, while bold lines mark the DKSCG. This graph contains the DCG which is disconnected since, in the absence of a full execution tracing, this information is partial. The LAR is shaded in light grey.

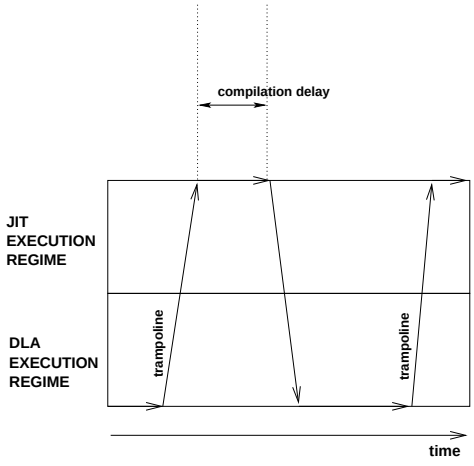

Fig. 6: DLA compilation falls into the worst application scenario (JIT) each time a trampoline is called.

correct prediction of the methods to compile in advance depends on the ability to trace the execution flow, we devote the remaining part of this Section to it.

3.2 Execution Trace Prediction

The correct prediction of the execution flow is required to keep the Look Ahead Region (LAR) correctly updated. It needs two kinds of information: the DKSCG, built at compile time, and the past execution trace, monitored at runtime.

The execution can be traced via *code instrumentation*, *asynchronous call stack sampling* [10] or *trampoline instrumentation*. Code instrumentation – e.g. at each method call – introduces an overhead, while asynchronous access to the call stack is required to be thread-safe, and must thus stall the execution.

On the other hand, trampoline instrumentation reduces the cost of tracing, but can lead to a loss of trace information since once a method is translated, its native address replaces the trampoline. In DLA compilation, this effect is amplified by the early compilation, which potentially replaces a large number of trampolines before their execution. This loss of information can be observed in Figure 5, which is highlighted by a disconnected known-DCG. Figure 6 shows the working of the trampolines. The execution of trampoline code means that the system is invoking a method not yet compiled, hence it is not working in an optimum DLA compilation regime due to bad execution trace prediction. Figure 7 shows this case, where bold lines represent methods invocations, and dotted lines represent the DKSCG.

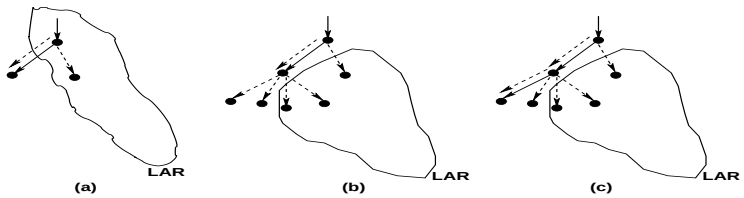

Fig. 7: Incorrect Execution Trace Prediction. (a) A method outside LAR is called through a trampoline. (b) LAR is updated, erroneously discarding two of the four children of the current method. (c) A method outside LAR – thus surely not yet compiled when invoked – is called through a trampoline.

In Figure 8, a good prediction leads to the compilation of several methods, but also to the loss of tracing information, as the removed trampolines cannot be exploited for execution tracing. The bold line encloses the compiled methods, while the LAR is shown in grey. The execution trace is represented by an arrowed line. This execution trace is unknown, as it always passes through native methods, without invoking trampolines. Moreover, since it enters into the LAR boundary, it is correctly predicted. But, due to the loss of information, LAR is not updated (Figure 8.b) and the execution exits the boundary (Figure 8.c), thus a trampoline is invoked.

When a trampoline is taken, the compilation overhead can be very large, since the just invoked method must wait for the compilation of all methods in the compilation queue. A two-queues prioritization mechanism can be used to reduce this delay, as

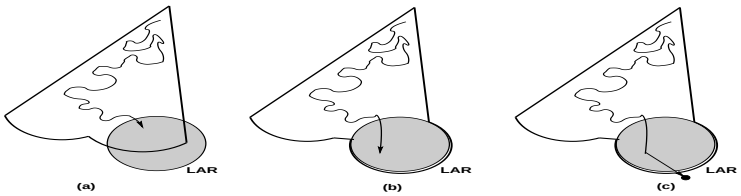

Fig. 8: Correct prediction with loss of tracing information. (a) (Unknown) execution trace stays inside the native methods region. (b) It enters into LAR, but the latter is not updated. (c) Trampoline call.

shown in the compilation framework of Figure 3. The invoked method is pushed into the high priority queue. The LAR is updated, and methods in the low priority queue, but not belonging to the new LAR, are dequeued.

Method Enqueueing Order Each time the LAR is updated, all new methods belonging to the compilation boundary are moved into the compilation queue. The enqueueing order is driven by the prediction model, which takes into account the likelihood of execution of each method in the next future. Static branch prediction techniques can help in building an accurate model [19, 9, 5].

If enqueueing order differs from the invocation order, the compilation overhead can be dramatic. If the executor invokes a method that is still into the compilation queue, the execution stalls until the method is dequeued and compiled.

In our DLA implementation, we consider two kinds of methods enqueueing order. The first is a simple FIFO ordering. The second exploits static branch prediction techniques [5] to compute the likelihood of each invocation (by setting parameters λ and δ of the model in Section 2). The LAR is updated on using a rough DKSCG distance based criterion in the first case, while in the latter this criterion is coupled with the likelihood of invocation. Section 4 provides an experimental evaluation, showing how a fine tuned model can lead to a better prediction.

3.3 Hot-Spots Detection

The effectiveness of DLA compilation in the long run depends on the ability to generate high quality native code for the application *hot spots*. The DLA compiler estimates whether a method could be a hot spot before compiling it. It computes the node weight w' described by the model of Section 2. Specifically, the hot spot detection affects the parameter t_{exc}, which measures the time complexity of a method. For this purpose, the DLA compiler analyzes the method structure and the DKSCG.

The former provides clues on its run-time behavior, e.g indicators are number of instructions, presence of computationally intensive loops. This information is partial, but can be enough for hot-spot detection. More detailed overviews of static method time complexity evaluation can be found in [1, 17]. For DKSCG contribution, consider the scenario shown in Figure 9, where the hot-spot marking is propagated through the DKSCG.

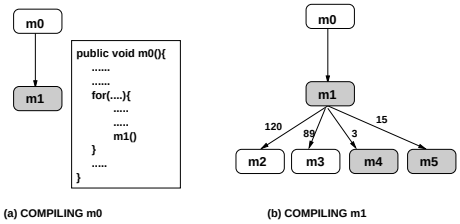

Fig. 9: Static hot spot detection based both on DKSCG and method structure. (a) While compiling `m0`, the DLA compiler discovers that it calls `m1` into a loop. `m1` is added to the DKSCG and marked as hot-spot. (b) While compiling `m1`, the DLA compiler marks as hot spots also its children that can be invoked through low weighted arcs.

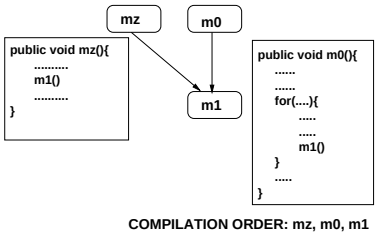

Fig. 10: Example of hot spot detection failure.

This approach, however, is not universally effective. Consider the example in Figure 10, where `m1` can be called both by `m0` and `mz`. In the latter case, the call is not within a loop. If `mz` is compiled before `m0`, then `m1` will not be marked as hot spot. When hot spot detection fails, a recompilation mechanism can be exploited. When a method is recognized as hot-spot it is pushed again into the compilation queue, even though previously compiled. This approach is similar to what described in [14].

4 Experimental Results

To give a first evaluation of typical performance improvements achieved by the DLA compilation we have considered two well known scientific benchsuite (`JavaGrande` [16] and `Scimark` [20]) as target. The DLA technique has been implemented into our dynamic compiler, called Intermediate Language Distributed Just In Time (ILDJIT) [7], briefly described by Figure 3. It has three different working modes: AOT, JIT and DLA. The target platform is an 8 processor Xeon at 2G-Hz, with 16GB of RAM and a $4MB$ cache for each pair of processors.

To show the benefits due to DLA compilation w.r.t. the standard JIT compilation, we have considered the ILDJIT JIT working mode as the baseline. We do not compare with other JIT compilers such as Mono or the Microsoft .NET Framework, since the goal of the experimental study is to evaluate the DLA technique rather than comparing different JIT compilers. The results of an experiment using different JIT compilers would be affected primarily by the differences in the quality of the generated code, thus making it more difficult to understand the impact of the DLA technique. However the ILDJIT compiler currently outperforms Mono with full optimization enabled by 3% on the set of benchmarks considered in this work.

To show the impact of the different choices in the abstract model parametrization, execution tracing technique, and prioritization of the compilation queues, we compare four versions of the DLA technique, shown in Table 1. In all versions, aggressive optimizations and hot-spot detection are used.

The two abstract models adopted use different definitions of the function f, which controls the enqueueing order (see Section 2):

Table 1: DLA implementations

Name	Model	Priority queue	Execution tracing
DLA1	M1	Yes	Trampolines
DLA2	M2	Yes	Trampolines
DLA3	M1	Yes	Execution Stack Sampling
DLA4	M2	Yes	Execution Stack Sampling

Table 2: Characterization of the full benchmarks in terms of methods defined, static call points and number of method invocations performed at runtime.

Benchmark	Methods defined	Call Points	Method invocations	Benchmark	Methods defined	Call Points	Method invocations
JGFArith	34	46	58	JGFFFT	58	1191	4191
JGFLoop	35	46	59	JGFSparseMatmult	54	1102	25094
JGFCast	34	46	58	JGFRayTracer	67	691	1678
JGFAssign	39	60	79	SciMarkSOR	47	2891	70267
JGFheapsort	54	67	35079	SciMarkMonteCarlo	45	4017	5600071
				SciMarkLU	55	10849	71367

M1 a *naive* implementation, where λ is dropped, δ is the order of appearance of a method in the parent body and $f(\frac{1}{\lambda}, \delta) = \delta$.

M2 a refined one, where λ and δ are estimated on the base of branch prediction analysis, as described in [5].

For both the models, a hot spot detector estimates the time complexity of the methods, t_{exc}, and the DKSCG node is weighted as $f'(t_{exc}) = t_{exc}$. Moreover, the LAR is updated following a fixed distance criterion over the DKSCG. We call this distance, the *boundary*. In model M2, this criterion is coupled with the information provided by the branch prediction technique; in this case, a method belongs to the LAR only if its distance is within the boundary *and* the branch prediction detects it as highly likely to be invoked.

Table 2 reports a characterization of the Java Grande and SciMark benchmarks in terms of methods defined and executed as well as of static call points. Since the DLA technique tries to compile methods before their invocation, the effectiveness of the prediction becomes more important when the number of invoked methods grows.

Table 3 reports the dynamic behavior of the different DLA approaches. The greater effectiveness of a well tuned prediction model can be explained in terms of the number of prioritized methods and taken trampolines. The lower these measures, the more precise the prediction of the execution flow. It means that the compilation threads are effectively able to provide in advance many native methods effectively executed in the near future.

Table 4 shows the execution time for several settings of the system. JIT and AOT compilers are provided with and without optimizations, JIT1 and JIT2 (AOT, respectively). Their performance are compared to DLA1, DLA2, DLA3 and DLA4. Three main considerations arise. First, DLA2 is always faster than DLA1, as well as DLA4 is faster than DLA3; this proves that a fine-tuning of the prediction model is significant for making the DLA compilation effective. Second, the higher the number of different invocable methods that make up the benchmark, the more important the execution tracing

Table 3: Dynamic execution characterization of JIT and DLA techniques. JIT1 and JIT2 (both reported as JIT) have the same behavior. AOT1 and AOT2 are neglected, since they have zeros for each column.

Benchmark	Compiler Technique	Methods translated	Trampolines taken	Methods prioritized	Classes analyzed	Benchmark	Compiler Technique	Methods translated	Trampolines taken	Methods prioritized	Classes analyzed
JGFArith	JIT	34	34	0	7	JGFSparseMatmult	JIT	54	54	0	16
	DLA1	48	43	14	15		DLA1	138	83	24	25
	DLA2	45	6	1	7		DLA2	108	4	4	18
	DLA3	47	43	31	14		DLA3	128	80	61	23
	DLA4	45	5	0	7		DLA4	108	3	3	18
JGFLoop	JIT	35	35	0	7	JGFRayTracer	JIT	67	67	0	52
	DLA1	55	37	13	13		DLA1	141	101	41	62
	DLA2	45	3	1	7		DLA2	121	11	7	54
	DLA3	49	37	29	12		DLA3	133	101	72	60
	DLA4	45	2	0	7		DLA4	119	9	5	54
JGFCast	JIT	34	34	0	7	SciMarkSOR	JIT	47	47	0	13
	DLA1	48	38	4	9		DLA1	69	51	13	17
	DLA2	45	4	1	7		DLA2	56	3	1	14
	DLA3	47	38	31	9		DLA3	66	51	42	16
	DLA4	45	4	1	7		DLA4	55	2	0	14
JGFAssign	JIT	39	39	0	13	SciMarkMonteCarlo	JIT	45	45	0	12
	DLA1	61	45	15	15		DLA1	69	46	10	18
	DLA2	52	3	1	13		DLA2	53	3	1	13
	DLA3	58	45	39	15		DLA3	66	46	37	17
	DLA4	52	2	0	13		DLA4	53	2	0	13
JGFheapsort	JIT	54	54	0	14	SciMarkLU	JIT	55	55	0	12
	DLA1	81	42	10	17		DLA1	81	51	25	16
	DLA2	64	8	3	14		DLA2	62	4	1	13
	DLA3	80	41	34	17		DLA3	78	51	43	16
	DLA4	64	6	1	14		DLA4	62	3	0	13
JGFFFT	JIT	58	58	0	18						
	DLA1	91	61	13	23						
	DLA2	74	9	4	20						
	DLA3	91	61	42	21						
	DLA4	72	7	2	20						

becomes. For these benchmarks, execution tracing efficiently drives DLA compilation. Hence, DLA3 and DLA4 translate fewer methods than DLA1 and DLA2. Finally, these results show how DLA compilation is a successful technique, which effectively bridges the gap between JIT and AOT compilation – often reaching an execution time close to that obtained executing a statically compiled code.

The following experimental results describes the LAR boundary impact on the DLA compilation, as well as the scaling of this technique w.r.t. the number of available CPUs.

Table 5 shows how more methods will be promoted for compilation in advance, when the boundary increases. Increasing the boundary, DLA1 scales worse than DLA2. The latter is able to determine – thanks to branch prediction – which methods are effectively to be pushed for compilation, choosing them from the large number of methods within the boundary. Moreover, execution tracing leads to a speedup when the benchmark has a sufficient number of methods, introducing overheads otherwise. In fact DLA3 and DLA4 outperform DLA1 and DLA2 only for JGFheapsort, JGFFFT, JGFSparseMatmult and JGFRayTracer.

Finally, Table 6 provides a characterization of DLA approaches as a function of the number of CPUs, taking into account DLA2 and DLA4 only. As expected, we can see that the DLA technique is only effective when multiple CPUs are available, and then only for benchmarks with a high number of methods. The performance scaling is not linear, and the performance quickly converges to an asymptote, as the number

Table 4: Java Grande and SciMark benchmarks: Execution time

Benchmark	Metric	JIT1	JIT2	AOT1	AOT2	DLA1	DLA2	DLA3	DLA4
JGFArith	Total time	171.96	145.72	171.5	127.05	141.15	129.15	141.291	129.171
	Machine code execution time	171.5	127.05	171.05	127.05	127.05	127.05	127.05	127.05
	Compilation delay	0.46	18.67	0	0	14.1	2.1	14.241	2.121
JGFLoop	Total time	6.774	5.211	6.136	3.644	4.454	4	4.462	4.003
	Machine code execution time	6.136	3.644	6.136	3.644	3.644	3.644	3.644	3.644
	Compilation delay	0.638	1.567	0	0	0.81	0.356	0.818	0.36
JGFCast	Total time	21.302	17.159	21.256	14.62	15.938	15.53	15.952	15.539
	Machine code execution time	21.256	14.62	21.256	14.62	14.62	14.62	14.62	14.62
	Compilation delay	0.046	2.539	0	0	1.318	0.91	1.331	0.919
JGFAssign	Total time	167.655	146.059	167.551	131.476	143.98	136.08	144.105	136.126
	Machine code execution time	167.551	131.47	167.551	131.476	131.47	131.47	131.47	131.47
	Compilation delay	0.104	14.589	0	0	12.51	4.61	12.635	4.656
JGFheapsort	Total time	58.022	56.696	57.943	53.303	55.922	54.213	55.896	54.204
	Machine code execution time	57.943	53.303	57.943	53.303	53.303	53.303	53.303	53.303
	Compilation delay	0.079	3.393	0	0	2.619	0.91	2.593	0.901
JGFFFT	Total time	66.561	61.671	65.294	54.983	59.032	56.943	58.951	56.904
	Machine code execution time	65.294	54.983	65.294	54.983	54.983	54.983	54.983	54.983
	Compilation delay	1.267	6.688	0	0	4.048	1.96	3.967	1.921
JGFSparseMatmult	Total time	17.439	13.374	16.714	8.487	12.397	9.617	12.319	9.594
	Machine code execution time	16.714	8.487	16.714	8.487	8.487	8.487	8.487	8.487
	Compilation delay	0.725	4.887	0	0	3.91	1.13	3.832	1.107
JGFRayTracer	Total time	51.91	45.535	50.981	28.62	37.53	31.23	37.263	31.152
	Machine code execution time	50.981	28.62	50.981	28.62	28.62	28.62	28.62	28.62
	Compilation delay	0.929	16.915	0	0	8.91	2.61	8.643	2.532
SciMarkSOR	Total time	61.342	58.24	61.25	49.12	55.27	51.93	55.332	51.958
	Machine code execution time	61.25	49.12	61.25	49.12	49.12	49.12	49.12	49.12
	Compilation delay	0.092	9.12	0	0	6.15	2.81	6.212	2.838
SciMarkMonteCarlo	Total time	39.3	23.303	39.25	16.222	20.601	18.322	20.645	18.343
	Machine code execution time	39.25	16.222	39.25	16.222	16.222	16.222	16.222	16.222
	Compilation delay	0.05	7.081	0	0	4.379	2.1	4.423	2.121
SciMarkLU	Total time	31.131	23.13	31.1	18.92	22.73	20.04	23.061	20.051
	Machine code execution time	31.1	18.92	31.1	18.92	18.92	18.92	18.92	18.92
	Compilation delay	0.031	4.21	0	0	3.81	1.12	4.141	1.131

of CPUs needed to perform the compilation steps is limited by the number of methods to compile. Thus, the scaling is expected to become more pronounced for large benchmarks, with many more methods. By the same token, we expected the difference between DLA2 and DLA4 to increase for larger benchmarks, as precision loss due to execution stack tracing would have a greater impact on the DLA performance.

Due to space constraints, we omit discussion about the negligible memory overhead due to DKSCG storage needed by the DLA compiler. We note, however, that the computation of DKSCG is performed using information that is required for the usual operation of the dynamic compiler, thus not resulting in significant performance overhead.

5 Related Works

A wide survey of *Just in Time* (JIT) and *Ahead of Time* (AOT) compilation can be found in [3] and [21].

Continuous Program Optimization [13] allows periodic code recompilation for adapting it to different workloads. In BEA's JRockit [6], methods are first compiled without

Table 5: DLA total execution time (in seconds) as a function of the maximum look-ahead distance from the executing method (Boundary).

	DLA1 Boundary					DLA2 Boundary					DLA3 Boundary					DLA4 Boundary				
Benchmark	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
JGFArith	141.15	144.15	143.966	143.941	143.941	140.56	137.15	135.46	129.15	129.15	141.291	144.321	144.135	144.11	144.11	140.695	137.251	135.544	129.171	129.171
JGFLoop	5	4.624	4.454	4.454	4.454	4.8	4.284	4	4	4	5.013	4.633	4.462	4.462	4.462	4.811	4.29	4.003	4.003	4.003
JGFCast	15.938	15.955	15.941	15.942	15.942	15.921	15.932	15.53	15.53	15.53	15.952	15.968	15.954	15.955	15.955	15.934	15.945	15.539	15.539	15.539
JGFAssign	145.59	145.28	143.98	145.62	145.62	141.78	138.62	136.08	136.08	136.08	145.731	145.418	144.105	145.762	145.762	141.883	138.692	136.126	136.126	136.126
JGFheapsort	55.922	56.696	56.684	56.513	56.543	55.922	55.915	55.821	54.613	54.213	55.896	56.662	56.65	56.481	56.511	55.896	55.889	55.796	54.6	54.204
JGFFFT	59.032	61.008	59.745	59.727	59.727	58.989	58.926	58.188	56.943	56.943	58.951	60.887	59.65	59.632	59.632	58.909	58.847	58.124	56.904	56.904
JGFSparseMatmult	13.197	12.637	12.497	12.397	12.397	13.167	12.497	11.697	9.637	9.617	13.103	12.554	12.417	12.319	12.319	13.073	12.417	11.633	9.614	9.594
JGFRayTracer	37.797	38.87	37.645	37.53	37.53	37.796	37.686	37.537	31.56	31.23	37.522	38.563	37.374	37.263	37.263	37.521	37.414	37.269	31.472	31.152
SciMarkSOR	58.13	55.27	57.22	56.94	56.94	57.24	52.24	51.93	51.93	51.93	58.22	55.332	57.301	57.018	57.018	57.321	52.271	51.958	51.958	51.958
SciMarkMonteCarlo	20.601	21.132	21.372	21.472	21.472	20.232	19.626	18.322	18.322	18.322	20.645	21.181	21.423	21.524	21.524	20.272	19.66	18.343	18.343	18.343
SciMarkLU	23.02	22.73	23.02	23.02	23.02	22.04	21.16	20.04	20.04	20.04	23.061	22.768	23.061	23.061	23.061	22.071	21.182	20.051	20.051	20.051

Table 6: DLA characterization over the number of CPUs; results are in seconds and they are the total execution time of the compiler

	JIT CPUs	DLA2 CPUs								DLA4 CPUs							
Benchmark	1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
JGFArith	171.96	182.684	137.147	129.214	129.15	129.15	129.15	129.15	129.15	191.604	138.745	130.8	129.171	129.171	129.171	129.171	129.171
JGFLoop	6.774	7.257	4.247	4.002	4	4	4	4	4	9.817	4.503	4.317	4.003	4.003	4.003	4.003	4.003
JGFCast	21.302	22.744	16.492	15.538	15.53	15.53	15.53	15.53	15.53	28.254	17.744	16.55	15.539	15.539	15.539	15.539	15.539
JGFAssign	167.655	183.761	150.826	136.197	136.08	136.08	136.08	136.08	136.08	200.291	152.082	137.396	136.126	136.126	136.126	136.126	136.126
JGFheapsort	58.022	134.438	57.497	54.902	54.268	54.213	54.213	54.213	54.213	146.998	57.372	54.79	54.266	54.204	54.204	54.204	54.204
JGFFFT	66.561	171.745	62.374	57.168	56.944	56.943	56.943	56.943	56.943	190.275	62.123	57.047	56.904	56.904	56.904	56.904	56.904
JGFSparseMatmult	17.439	28.521	13.339	9.647	9.617	9.617	9.617	9.617	9.617	37.721	13.214	9.635	9.594	9.594	9.594	9.594	9.594
JGFRayTracer	51.91	161.373	50.607	40.803	37.476	36.851	35.915	34.353	31.23	174.688	49.351	39.878	37.376	36.839	35.894	33.432	31.152
SciMarkSOR	61.342	89.431	57.188	52.013	51.93	51.93	51.93	51.93	51.93	96.851	57.176	52.013	51.958	51.958	51.958	51.958	51.958
SciMarkMonteCarlo	39.3	42.28	20.875	18.342	18.322	18.322	18.322	18.322	18.322	45.53	20.851	18.343	18.343	18.343	18.343	18.343	18.343
SciMarkLU	31.131	35.797	23.142	20.065	20.04	20.04	20.04	20.04	20.04	37.953	23.127	20.065	20.051	20.051	20.051	20.051	20.051

optimizations. A single thread is used both for compilation and execution, while a parallel one samples the execution and triggers aggressive recompilation of "hot" methods. While this paper focuses on the DLA technique itself, continuous optimization is just as easily implemented in a DLA compiler as in a traditional JIT. A method that needs recompilation is treated as a new method by the DLA compiler.

Selective Compilation is used to minimize compilation overheads while still achieving the largest part of the beneficial effects of JIT compilation. The Sun Microsystems Java HotSpot Virtual Machine [24] runs both an interpreter and a compiler, the latter invoked on hot-spots [18]. In [1], an evaluation of several techniques for *hot spot* detection is presented. The main difference between DLA and Selective Compilation is that the former aims at predicting in advance which methods are hot spot and which not, both hiding the compilation time and ensuring good quality binary code. Moreover, DLA compilation is based on prediction techniques that analyzes the static code properties, even though they are applied dynamically; conversely, selective compilation is mainly based on dynamic profiling, which requires code instrumentation. However, the two techniques could be adapted to work together.

Adaptive Optimization merges *Continuous Program Optimization* and *Selective Compilation*. A complete survey can be found in [2], while further considerations are presented by Kulkarny *et al.* [15]. This approach exploits a dedicated thread to detect hot spots and optimize them. The optimizer thread is run asynchronously w.r.t. the execution flow. In a multiprocessor environment, the optimization time can be masked.

Background Compilation [14] is directly related to DLA compilation. Optimization is performed on dedicated hardware, on the base of an off-line profiling phase. If a

method still lies into the optimization queue at its invocation, lazy compilation is employed. This is the main difference w.r.t. DLA compilation. However, these techniques could be coupled since DLA compilation is orthogonal w.r.t dynamic code profiling. A more distantly related approach has been proposed in [12], involving the use of a compilation thread to guarantee an upper bound to the occupation of processor by the compiler by means of earliest deadline first scheduling.

Another work partially matching the DLA compilation is presented by Unnikrishnan *et al.* in [25]. Multiple threads on multiple processors re-compile and optimize in advance those code portions with high chances to be executed soon or requiring further improvements. The main difference w.r.t DLA compilation is that two kinds of run-time information are required in this case: the sampling of the execution trace and the profiling of properties such as time or energy consumption. Code instrumentation is needed to collect this information, which would impact the performance. Moreover, a method is only optimized after it has been invoked several times.

To be really effective, the DLA compilation has to follow a precise prediction model, described in Section 2 and based on the assignment of merit factors to the vertices and the arcs of a call-graph portion. These are computed in the *Look Ahead* phase on the base of structural parameters. State of the art branch prediction techniques are described in [19, 9, 5]. An approach for fast hot spot estimation/detection is reported in [1]; another method-time-complexity evaluation can be found in [17]. Sophisticated techniques for optimization profit estimation are described in [26] and [4].

Last but not least, some considerations on polymorphism in OO languages are needed, since we claim that DLA compilation is most effective for those applications characterized by large static and dynamic call graphs, in terms of number of methods. These call graphs are typical of OO application, as also underlined in the DaCapo[1] benchmark suite [11]. Polymorphism introduces an uncertainty in method naming, thus making the run-time alias analysis a costly but effective optimization, as pointed in [22, 8]. On-line method versioning explicit the uncertain on the method to invoke producing several versions of the same method.

6 Conclusions

We have introduced the DLA compilation technique, covering theoretical and practical problems related to it and showing how DLA compilation can be a powerful technique to reduce the impact of dynamic compilation time and to generate high quality native code. Its effectiveness proves to be strictly related to the ability both to correctly predict the execution flow and to apply the right set of optimizations to each code region. A more precise matching of the DLA compiled methods set to the execution trace directly results into a reduction of the compilation delay, while the optimization of the hot spots should guarantee the execution of good quality code. On the Java Grande and SciMark benchmark set (not a favourable one, since it has a reduced use of polymorphism) we obtain an average speedup of 15%, and an average reduction of the overhead to less than 1% of the JIT compilation time. It is to be expected that the benefits of DLA compilation

[1] We do not take DaCapo benchmarks into consideration in the experimental evaluation because ILDJIT does not support generics at this time.

will be higher for applications characterized by large call graphs, which are typical of Object Oriented highly polymorphic applications. This will be investigated in the future.

References

1. G. Agosta, S. Crespi Reghizzi, P. Palumbo, and M. Sykora. Selective compilation via fast code analysis and bytecode tracing. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 906–911, New York, NY, USA, 2006. ACM.
2. M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, Feb 2005.
3. J. Aycock. A Brief History of Just-In-Time. *ACM Comp. Surveys*, 35(2):97–113, Jun 2003.
4. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
5. T. Ball and J. R. Larus. Branch Prediction For Free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.
6. BEA JRockit: Java for the enterprise technical white paper, 2006.
7. S. Campanoni, G. Agosta, and S. Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.
8. J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.
9. B. L. Deitrich, B.-C. Cheng, and W.-M. W. Hwu. Improving Static Branch Prediction in a Compiler. In *IEEE PACT*, pages 214–221, 1998.
10. Michael Dunlavey. Performance tuning with instruction-level cost derived from call-stack sampling. *SIGPLAN Not.*, 42(8):4–8, 2007.
11. S. M. Blackburn *et al.* The DaCapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
12. Timothy Harris. Controlling run-time compilation. In *In Procedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 75–84, 1998.
13. T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.
14. C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software Practice and Experience*, 31(8):717–738, 2001.
15. P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE*, pages 94–104, 2007.
16. 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.
17. D. Le Métayer. ACE: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.
18. M. Paleczny, C. A. Vick, and C. Click. The Java HotSpot Server Compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001.
19. J. R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 67–78, 1995.
20. R. Pozo and B. Miller. http://math.nist.gov/scimark2. SciMark benchmark.
21. T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler. In *Proc. of the Third Conference on Object-Oriented Technologies and Systems*, Jun 1997.
22. D. Rayside. Polymorphism is a Problem. In *Panel on Reverse Engineering and Architecture, CSMR'02*, Mar 2002.
23. Kazuyuki Shudo. Performance comparison of java/.net runtimes. http://www.shudo.net/jit/perf, 2005.
24. Sun Microsystems Java team. The Java HotSpot Virtual Machine, v1.4.1.
25. P. Unnikrishnan, M. Kandemir, and F. Li. Reducing dynamic compilation overhead by overlapping compilation and execution. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 929–934, Piscataway, NJ, USA, 2006. IEEE.
26. M. Zhao, B. R. Childers, and M. L. Soffa. An approach toward profit-driven optimization. *ACM Trans. Archit. Code Optim.*, 3(3):231–262, 2006.