Static Memory Management within Bytecode Languages on Multicore Systems

Simone Campanoni
Harvard University
33 Oxford street
02138, Cambridge, USA
Email: xan@eecs.harvard.edu

Luca Rocchini
Dipartimento di Elettronica e Informazione
Politecnico di Milano
34/5 Ponzio
20133, Milan, Italy
Email: rocchini@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 pipeline model is appealing for the compilation tasks that dynamic compilers need to perform, but it can bring deadlock issues when static memories are exploited by the running program. This research suggests a solution that both solves the mentioned problem and reduces the unnecessary compiler threads used to handle static memories. The proposed solution is a self-aware runtime system that both it is able to detect/avoid deadlocks and it adapts the number of compilation threads needed to handle static memories to the current workload.**

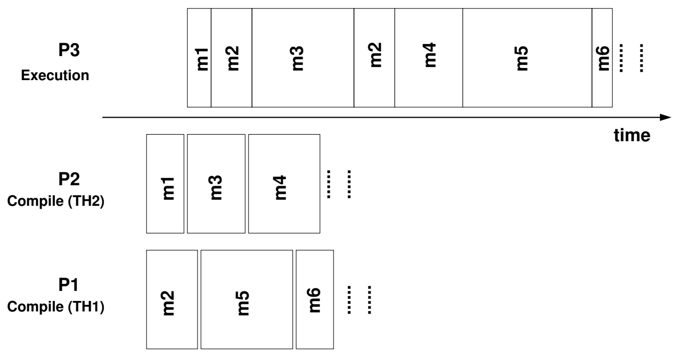

Fig. 1. An ideal case. Each invoked method has already been compiled and optimized.

I. Introduction

Software portability suggests the generation of portable intermediate binary code, that remains independent from the specific hardware architecture and is executed by a software layer called *virtual machine (VM)*. A virtual machine provides an interface to an abstract computing machine that accepts the intermediate binary code as its native language; in this way, the virtualization of the Instruction Set Architecture (ISA) is performed.

The first generation of VMs was entirely interpreted: they interpreted byte-code [6], [3], [10], [12] rather than compiling it to machine code and executing directly the machine code. This approach, of course, did not offer the best possible performance, as the system spent more time executing the interpreter than the program it was supposed to be running. These first VMs quickly fell into disuse for their slowness.

The next generation of VMs used just-in-time (JIT) [19], [2], [15], [1] compilers to speed up the execution, by exploiting more memory resources. Strictly defined, a JIT compiler translates byte-code into machine code, before the execution, in a lazy fashion: the JIT compiles a code path only when it knows that code path is about to be executed (hence the name, just-in-time compilation). This approach allows the program to start up more quickly, as a lengthy compilation phase is not needed before execution beginning. The JIT approach seemed promising, but it presented some drawbacks: JIT compilation removes the overhead due to the interpretation at the expense of some additional startup cost, and the level of code optimization is mediocre. To avoid a significant startup penalty for portable applications, the JIT compiler has to be fast, which means that it cannot spend much time in optimization. For this reasons, researches proposed several approaches to find a good trade-off between the time spent to compile the code and time spent executing it [18], [8]. Moreover, the performance of JIT compilers is still lower than that of native code produced by static byte-code compilation [17], or *Ahead Of Time* (AOT) compilation.

After JIT compilers became widely used, multi-core technology emerged in high-performance architectures. Such architectures need specifically designed multi-threaded software to exploit all the potentialities of their hardware parallelism.

Recently, a new generation of virtual machines, called *Dynamic Look-Ahead (DLA)* compilers [5], has been introduced. These dynamic compilers rely on a software pipeline architecture for compilation, optimization and execution tasks. This speeds up the execution by exploiting the parallelism provided by the multi-core technology to produce optimized

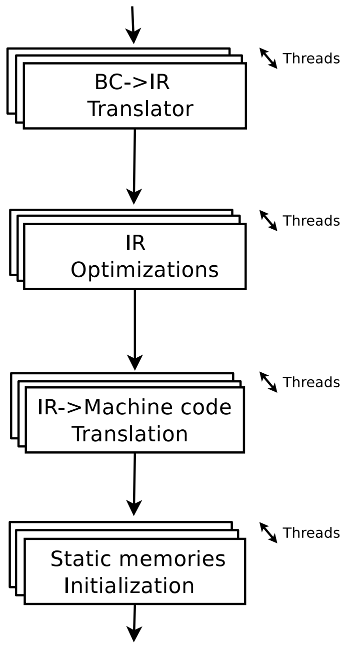

Fig. 2. Compilation pipeline model. The languages used within the pipeline are: *BC* - the input bytecode language of the compiler, *IR* - the intermediate representation used by the compiler and *Machine code*, which depends on the underlying platform.

code. Strictly defined, a DLA compiler translates and optimizes byte-code looking ahead of the execution in order to both anticipate the compilation before the execution asks for it, and to produce optimized code. As [5], we consider a multiprocessor environment where in the best case scenario, there is no compilation overhead, because compilation fully overlaps with execution and methods are already compiled when they are invoked (see Figure 1).

The pipeline model is appealing to carry out compilation tasks that dynamic compilers, like DLA, need to perform, but it can bring deadlock issues when static memories are used by the running program. This paper proposes a solution to this problem, which is also able to reduce unnecessary compiler threads.

The rest of the paper is organized as follows. Section II describes the pipeline execution model, which targets multi-core architectures. This section describes also DLA compilers, which heavily exploit this model. After the description of the problem of initializing static memories, Section III introduces both an already introduced solution for single-threaded compilers and the solution proposed in this paper, which is suitable for compilers that rely on the pipeline model. Section IV provides details on the evaluation of the solution proposed. Finally, Section V concludes the paper.

II. Pipeline Execution Model

The pipeline model exploits the parallelism available across compilation tasks within both dynamic and static compilers [5], [4].

The delay spent within each compilation stage depends on characteristics of the specific method that we are considering (e.g. number of instructions, Control-Flow-Graph, etc. . .); for this reason, the order of the methods that enter to the pipeline may differ w.r.t. the output order. This effect has to be considered both to avoid deadlock within the compilation pipeline and to ensure that a method leaves the pipeline only when it is ready to be executed (i.e., both available in machine code and the static memories used by it have been initialized).

As shown in Figure 2, there are 4 stages within the compilation pipeline; the first translate a method from the input bytecode (e.g. Java, CIL, etc. . .) to the intermediate representation used by the compiler, say IR. The second stage optimizes the code by exploiting the language IR (i.e., translation from IR to IR). The third stage translates a method to the machine code of the underlying platform (e.g. Intel x86, ARM, etc. . .). Finally, the last stage initializes the static memories that can be read or write (used) by the method in the pipeline.

Every stage of the pipeline is composed by a set of parallel threads, which are hereafter called *compiler threads*. Moreover, compiler threads that compose the last stage of the pipeline are called *static threads* because they are in charge to initialize static memories (see Section III).

An example of usage of the pipeline model within dynamic compilers is the Dynamic Look-Ahead compilation (DLA) [5], which is following described.

A. Dynamic Look-Ahead compilation

Dynamic Look Ahead (DLA) [5] is a recent compilation technique, which aims to mask as much as possible the compilation delay by overlapping the compilation efforts. As Figure 3 shows, this technique predicts at runtime where the execution is going to go and it starts the compilation of parts of the application code before the execution asks for it.

This model is based upon the use of two priority queues (see Figure 4) (a low priority one and a high priority one) and on runtime code profiling of the bytecode application. 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.

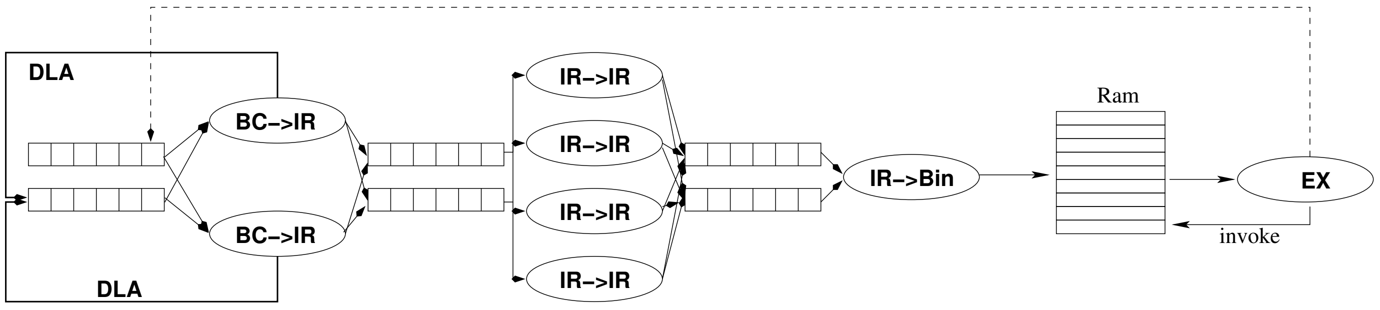

Fig. 4. Internal organization of a DLA compiler. BC is the source bytecode (CIL) and ovals are the internal threads of the compiler

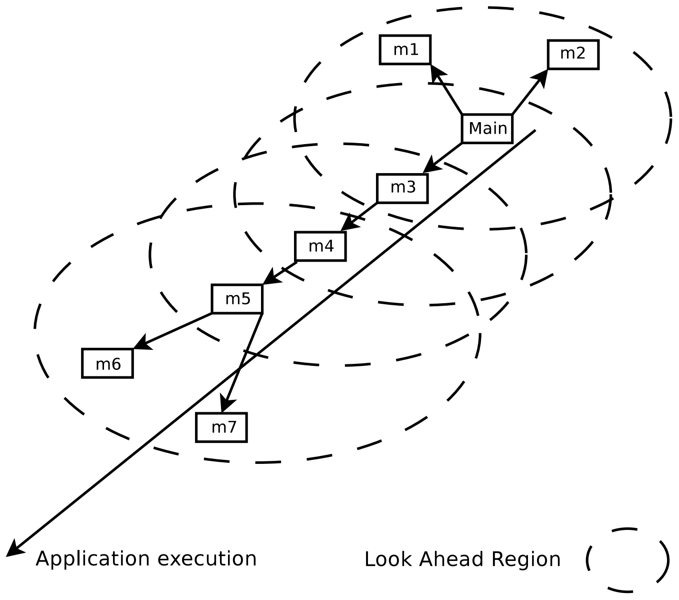

Fig. 3. Look-Ahead-Region of DLA compilers

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.

III. Handling Static Memory

Previous sections describe how the pipeline model can be used within dynamic compilers in order to better schedule compilation tasks on multiprocessor platforms. With or without a pipeline model, handling static memories can bring deadlock issues (see Section III-A).

In order to initialize static memories, bytecode systems like the Common Language Infrastructure (CLI), which includes the CIL bytecode [9], rely on special methods; those need to be called by the runtime system (i.e., the dynamic compiler). Hence, the dynamic compiler is in charge to execute these special methods before the first use of their associated memories, even if there is no explicit call to them within the bytecode program.

The standard that describes the bytecode system provides the rules to follow to identify these special methods. For example, ECMA-335 [9] standard sets the names of these methods equal to "*cctor*". Hence, the method that is in charge to initialize the static memory of a class *A* is *A.cctor*. This paper assumes that for every method m, there is a set of methods M (which can be empty) that have to be compiled and executed in order to initialize the static memories that can be read or written by m.

The solution proposed by this paper to the problem of handling static memories is composed by two self-aware components: a runtime ables to detect and avoid deadlocks and a new algorithm that adapts the number of static threads based on the runtime workload.

After the description of the deadlock problem (see Section III-A) and its already proposed solution for single-threaded compilers [9] (see Section III-B), this section introduces our new approach, which targets multi-threaded compilers (see Section III-C).

A. Deadlock issues

The following example highlights the deadlock problem brought by handling static memories. Let s_1 and s_2 be two static memories, and let they need each other. In bytecode systems, if there is a dependences cycle like the one considered,

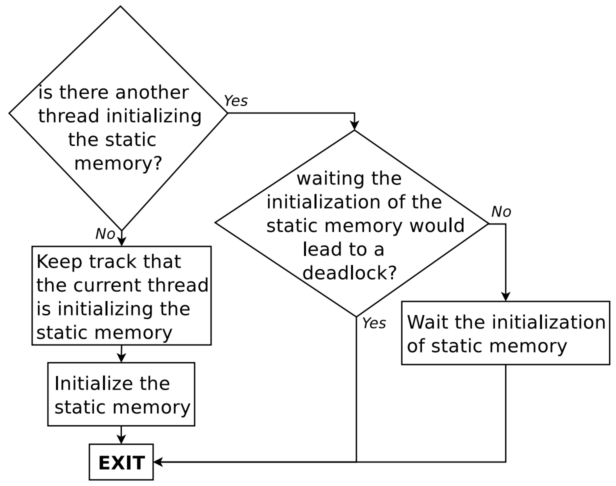
is there another thread initializing the static memory?
Yes
No
waiting the initialization of the static memory would lead to a deadlock?
No
Yes
Keep track that the current thread is initializing the static memory
Wait the initialization of static memory
Initialize the static memory
EXIT

Fig. 5. Algorithm used to initialize static memories without having a compilation pipeline.

the language guarantees correctness by letting to the compiler the freedom to choose the first memory to handle from this cycle. In our example, the compiler can start initializing s_1 or s_2. For this reason, whenever the compiler is composed by one thread, a simple algorithm is able to fix deadlock issues by checking these dependences cycles (see Section III-B).

Let us consider the multi-threaded case when the pipeline model is used. Consider a compiler thread t_1 that is going to initialize s_1; t_1 locks s_1 and it requests the compilation of the necessary methods, say M_1. Eventually, this compilation leads to the initialization of s_2 (due to the dependence between s_1 and s_2), which is performed by another thread, say t_2 (due to the pipeline model). t_2 requests the compilation of the necessary methods, say M_2; eventually, this compilation leads to the initialization of s_1 (due to the dependence between s_2 and s_1), which leads to a deadlock. Notice that t_1 cannot mark s_1 as "initialized" when it requests the compilation of M_1.

B. Single-threaded compilers

In this case, there is no compilation pipeline and the thread that requests the compilation of a method m is the same one that effectively performs the compilation, initializes the static memories used by m and restarts the execution of the produced code. Notice that even if the compiler is not composed by different threads, the bytecode program can be; hence, a not trivial algorithm to avoid deadlocks is necessary.

Figure 5 shows the solution proposed in [9] to initialize static memories for single-threaded compilers, which is following described. In the case a static memory is not marked as "initializing" (i.e., there is no other thread that is initializing this memory), the current thread marks it before doing it effectively. On the other hand, if this memory has been marked, (i.e., someone is already initializing this memory), an additional check is needed in order to detect a possible deadlock (by checking if there is a dependences cycle, see Section III-A). If there is no possibility of deadlocks, the current thread waits the initialization of the memory performed by the other one; otherwise, it returns to the execution of the program.

C. Multi-threaded compilers

A new solution is necessary to handle static memories for multi-threaded dynamic compilers that exploit the pipeline model described in Section II. The pipeline model makes the identification of deadlocks harder w.r.t. single-threaded compilers.

This section starts by describing the algorithms implemented within the compilation pipeline needed to ensure that static memories are initialized before their first use. Finally, this section provides information about how the compiler can adapt the number of threads based on the current workload (e.g. how to increase, decrease, the number of threads needed to initialize static memories). Increasing and decreasing this number is necessary in order to avoid deadlocks: every time there is no free static threads in the pipeline, new ones have to be created (see Section III-C2 for further details).

1) Algorithms: The compilation pipeline provides two types of methods compilation: synchronous and asynchronous compilation. Synchronous compilations return to the callee as soon as the input method is compiled (i.e., ready to be executed). On the other hand, the asynchronous version returns to the callee as soon as the input method is inserted to the compilation pipeline.

Both compilation types require new algorithms to handle static memories. These algorithms are introduced both within pipeline stages and within insertion routines (routines in charge to insert methods to the compilation pipeline). These algorithms exploit a directed graph called *waiting graph* W in order to keep track of both which thread is waiting for which method (edges from a thread t to a method m) and which thread is in charge to execute which method (edges from a method m to a thread t). The waiting graph W is used in order to detect, and therefore avoid, deadlocks as following described.

Synchronous insertion of methods. There are two types of threads that can insert methods to the compilation pipelines: the ones that are in charge to execute the code dynamically produced and static threads (see Section II).

When a method m is inserted, the type of the thread of the callee is checked as described in Figure 6: if this thread is not static, the insertion of m is provided in the obvious way: m is inserted and the thread yields the processor till both m is available in machine code and the necessary static memories have been initialized (i.e., m is ready to be executed). On the other hand, if the callee is a static thread, then m is checked if it is already within the pipeline or not. If m was not already inserted, the information that the callee thread needs m is inserted within the waiting graph W, m is inserted to the pipeline, the number of threads available to initialize static memories is decreased and the callee thread yields the processor till m will be ready to be executed. As soon as m becomes executable, the callee thread is waked up, it increases the number of static threads and it removes the information

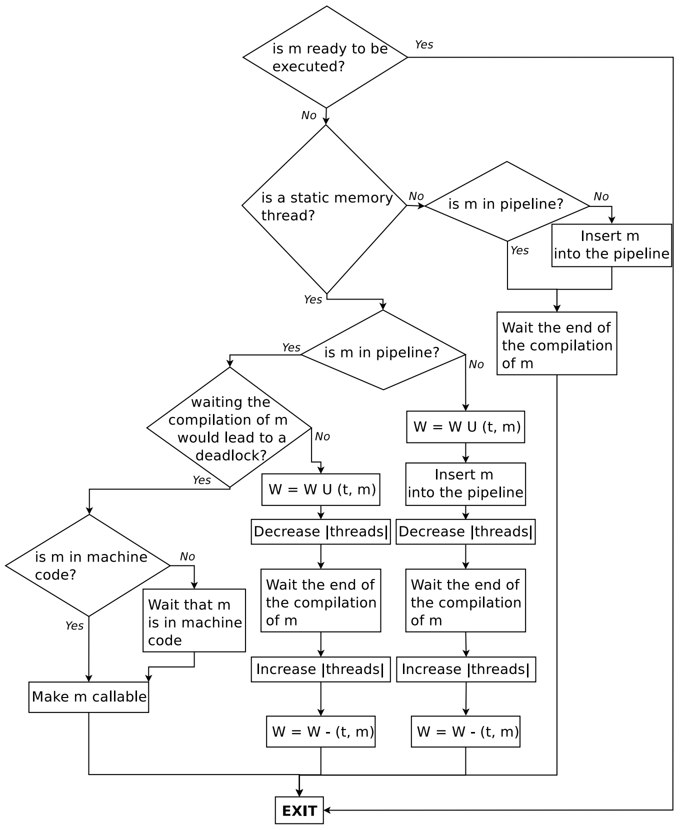
is m ready to be executed?
Yes
No
is a static memory thread?
No
is m in pipeline?
No
Insert m into the pipeline
Yes
Yes
Wait the end of the compilation of m
is m in pipeline?
Yes
No
waiting the compilation of m would lead to a deadlock?
No
Yes
W = W U (t, m)
Insert m into the pipeline
W = W U (t, m)
Decrease |threads|
Decrease |threads|
is m in machine code?
No
Yes
Wait that m is in machine code
Wait the end of the compilation of m
Wait the end of the compilation of m
Increase |threads|
Increase |threads|
Make m callable
W = W - (t, m)
W = W - (t, m)
EXIT

Fig. 6. Algorithm used to decide what actions should be performed in order to satisfy a request to insert a method into the compilation pipeline synchronously.

that it is waiting for the method m from W. On the other hand, if m was already inserted to the pipeline, then it means that some thread is translating it because m is not yet ready to be executed (check performed at the beginning of the algorithm, see Figure 6). For this reason, the callee thread cannot insert m to the compilation pipeline; instead, it waits till m will be ready to be executed.

Waiting m could lead to a deadlock in some situations like the following one: a static thread t_i is executing the code to initialize the memory used by the method m_i and this execution is stopped to wait the compilation of another method, say m_j (which is therefore inserted synchronously to the compilation pipeline). When m_j reaches the last stage of the pipeline (t_i is still waiting for it), another static thread, say t_j, starts running the code to initialize the static memory used by m_j, which leads to the method m_i. The static thread t_j tries to insert m_i to the pipeline, which is still inside, and if it would wait for it, there will be a deadlock (i.e., t_i is waiting for t_j and vice versa).

For this possibility of deadlocks, a check is performed within the graph W: if by inserting the edge (t, m) a cycle is created, then the thread t cannot wait for m; instead, the thread t waits only that m is available in machine code (i.e., it waits till m leaves the translation and optimization stages), it makes m callable by replacing its trampolines [14] and it

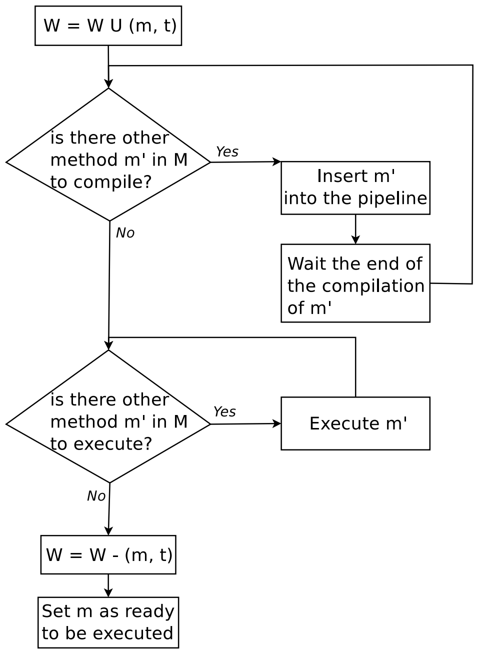
W = W U (m, t)
is there other method m' in M to compile?
Yes
Insert m' into the pipeline
Wait the end of the compilation of m'
No
is there other method m' in M to execute?
Yes
Execute m'
No
W = W - (m, t)
Set m as ready to be executed

Fig. 7. Algorithm used within the last pipeline stage.

leaves the pipeline.

On the other hand, if there is no risk of deadlocks, the current thread adds (t, m) to the waiting graph W, it decreases the number of static threads, it waits the compilation of m and it increases back this number, and also it removes (t, m) from W, before leaving the pipeline.

Asynchronous insertion of methods. In order to start the compilation of some methods before the execution asks for it (see Section II-A), an asynchronous version of the insertion procedure is provided. Whenever a method m is inserted, the routine checks if m is ready to be executed and, if this is the case, it returns to the callee. On the other hand, if m needs to be compiled, it is inserted to the compilation pipeline and the control is returned back to the callee.

Translate and optimize the code. Translations from the input bytecode (BC) to the intermediate representation (IR), from IR to the machine code as well as code optimizations are handled by getting the method m from the upstream pipeline stage, by performing the specific operation and by inserting m to the downstream pipeline stage.

Initialize static memories. The last stage of the pipeline is handled in a different way w.r.t. the first three stages (as shown in Figure 7). First the information that the method m is going to be translated by the current static thread t is recorded by adding the edge (m, t) to the waiting graph W.

Remember that for any method m there is a set of methods M to be executed in order to initialize static memories used by m (see Section II); hence, methods in M have to be compiled and executed before letting m leaving the pipeline. The thread t inserts synchronously to the pipeline every method $m' \in M$. When methods in M are ready to be executed, t starts executing them once per time.

When t has compiled and executed every method $m' \in M$, it removes the edge (m, t) from the waiting graph W and it sets m as *ready to be executed* before removing it from the compilation pipeline.

2) Adapt the threads number: This section describes how to adapt the number of static threads to both avoid deadlocks and improve the performance.

Avoid deadlocks. The number of static threads needs to be unbounded for the following described problem. By contradiction, let n_t be the bound of the number of static threads. Consider the case of a set of static memories $S = \{C_1, \ldots, C_{|S|}\}$, where $|S|$ is greater than the number n_t. Assume that for all memories $C_i \in \{C_1, \ldots, C_{|S|-1}\}$, C_i needs C_{i+1} for its initialization and m_i is the method that needs to be executed to initialize C_i. Then, if C_1 is the first memory to be initialized, thread T_1 (assuming the whole compilation pipeline is empty at the beginning of the operation) is in charge to compile and execute m_1 and therefore it inserts the method into the pipeline waiting for its compilation. Consequently, each thread T_i, $i \in [0, n_t)$ will be used to held the method m_i needed by the initialization of C_i, and the whole system will exhaust the set of available threads, but the method m_{n_t} for C_{n_t} will need the compilation of the method m_{n_t+1}, since $n_t < |S|$. In this case, there is no more static threads and the whole system goes into a deadlock.

Performance. The compiler can keep growing the number of static threads by adding new ones whenever is necessary (this case is referred as "baseline" in Section IV). The problem of this approach is that whenever the total number of compiler threads is bigger than the number of cores of the underlying platform (which is likely to be the case), the threads scheduler can play a critical role for the performance of the system. More threads are within a system, more critical is the OS thread scheduler. For this reason, the compiler should both keep enough threads to handle static memories and remove threads that are not necessary anymore.

This paper proposes a solution (referred as "policy" in Section IV) where the number of new created threads grows exponentially every time there is no available static threads. Moreover, a new thread is introduced within the compiler, which wakes up every time the compilation pipeline is empty and it reduces the number of static threads to 2: one for the high priority compilation queue and one for low priority one (see Section II-A).

IV. Experimental evaluation

We evaluated the solution proposed in this paper on a commercial Intel-based system by extending the dynamic compiler available within ILDJIT [4] compilation framework. This dynamic compiler generates machine code for applications written in CIL [9] bytecode. To translate benchmarks from C to CIL, we used the static compiler GCC4CLI [7].

After describing the hardware platform (Section IV-A) and the benchmarks (Section IV-B) used to evaluate the proposed approach, Section IV-C provides the obtained results.

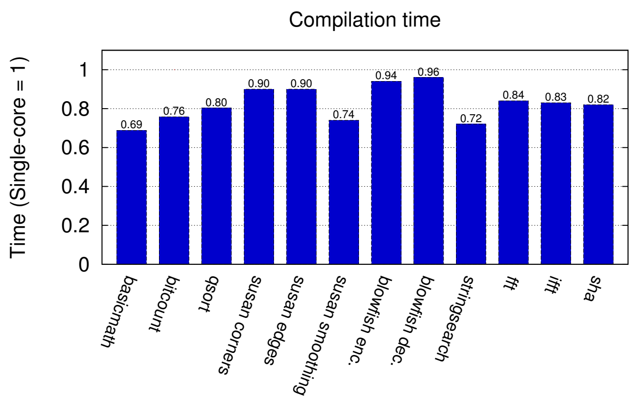

Fig. 8. Compilation time spent by the compiler with the compilation pipeline relative to the case where the compilation is performed by a single thread (no pipeline is used in this case).

A. Hardware Platform

To run our experiments we used an Intel® Core^{TM} i7 with 6 cores, each operating at 3.33 GHz. The processor has three cache levels. The first two are private to each core and are 32KB and 256KB each. The last level is 12MB and is shared between all cores [16]. In our evaluation, both Turbo Boost and Hyper-Threading were disabled.

B. Benchmarks

To evaluate our scheme we used applications from the CBench benchmark suite [11], which is based on MiBench [13]. While we attempted to consider the full spread of workloads from the suite, we had to discard those that failed to compile correctly under GCC4CLI [7]. We selected the largest input available for each application to exercise the benchmarks fully.

C. Evaluation

We evaluate the proposed approach by comparing the compilation time spent with and without the pipeline model implemented as described in [5] with the solution proposed in this paper. Moreover, we provide the information about the number of static threads allocated with and without the policy described in Section III-C2.

Compilation time. Here we provide the time spent by the dynamic compiler on blocking the execution of the program given as input in order to compile the code (also called *perceived compilation time*). We compare this time in two cases: when the compilation pipeline is used and when the compilation is performed by a single thread. We focus the attention to the compilation time because it does not depend on input data (as opposite for the time spent running the produced code).

Figure 8 shows the compilation time reduced by introducing the pipeline model to the dynamic compiler ILDJIT when it runs on top of the 6-cores machines described in Section IV-A.

As expected, for almost every benchmark, the compilation time is reduced thanks to the compilation pipeline and the solution proposed in this paper, which avoid deadlocks and reduces the proliferation of unuseful threads (next described). On the other hand, for susan corners, susan edges, blowfish enc. and blowfish dec. the compilation time is reduced only by 10%, or even less. This is due to the code analysis performed by the compilation pipeline (see [5] for further details), which predicts wrongly where the execution is going to go and therefore the order of methods compiled does not match the execution order, which reduces the utility of having a compilation pipeline.

Static threads. Here we provide the results achieved by the policy described in Section III-C2, which changes the number of static threads during the execution of the program. Figure 9 proves the importance of having a manager for these threads by showing the difference on having or not the mentioned policy. In this case, we consider the overall execution of the dynamic compiler (i.e., time spent to both compile and execute the code).

We can notice that most of the benchmarks need static threads only during the first 6% of the overall execution. The only exceptions are blowfish enc. and blowfish dec., which require compilation efforts till the first 32% of the program execution. This suggest that is not a good design choice to keep every static thread alive for the entire life time of the program execution. For this reason, the policy suggested in this paper (see Section III-C2) minimizes the number of static threads every time the compilation pipeline is empty. This approach gives the ability to the dynamic compiler to remove static threads when compilation tasks are not necessary anymore (after the first part of the program execution), or when they are not currently needed (within the first part of the program execution).

Considering the behavior described in both Figure 9(a) and Figure 9(c), we can notice that when there is a burst of usage of static threads, our solution adapts the runtime, by growing the number of static threads, faster than the baseline producing therefore a better reactive system. This faster adaptation is due to the policy used, which increases the number of new created threads exponentially (see Section III-C2).

V. Conclusion

Dynamic compilation is often used for object-code virtualization. The increasing availability of multiple processors on hardware platforms suggests their exploitation to compile code in parallel with the execution. The pipeline model is used for this target within DLA compilers. This model can bring deadlock issues when static memories are exploited by the running program. This paper proposes a solution that makes the compiler a self-aware system ables to detect and avoid deadlocks. Moreover, results achieved show that this approach reduces significantly the unnecessary compiler threads used to handle static memories.

Acknowledgments

Authors would like to thank Prof. Stefano Crespi Reghizzi for his non-stop supporting to the whole ILDJIT team, Dr. Aaron Smith and Prof. Marco D. Santambrogio for their support to the ILDJIT project. Additionally, authors thank the anonymous reviewers for their insightful comments. This material is based upon work supported by both Microsoft and HiPEAC. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of Microsoft.

References

[1] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35:97–113, June 2003.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[3] J. R. Bell. Threaded code. *Commun. ACM*, 16:370–372, June 1973.

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

[5] S. Campanoni, M. Sykora, G. Agosta, and S. Crespi-Reghizzi. Dynamic Look Ahead Compilation: A Technique to Hide JIT Compilation Latencies in Multicore Environment. In *CC*, pages 220–235, 2009.

[6] K. Casey, M. A. Ertl, and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29, October 2007.

[7] R. Costa, A. Ornstein, and E. Rohou. http://gcc.gnu.org/projects/cli.html. GCC4CLI.

[8] V. C. de Verdiere, S. Cros, C. Fabre, R. Guider, and S. Yovine. Speedup prediction for selective compilation of embedded java programs. In *In Proc. of EMSOFT*, 2002.

[9] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. Fourth edition, June 2006.

[10] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32:265–294, March 2002.

[11] G. Fursin. Collective Tuning Initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009.

[12] E. Gagnon. *A portable research framework for the execution of java bytecode*. PhD thesis, Montreal, Que., Canada, Canada, 2003. AAINQ88471.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[14] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

[15] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[16] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *PACT*, 2009.

[17] K. Shudo. Performance comparison of java/.net runtimes. http://www.shudo.net/jit/perf, 2005.

[18] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM.

[19] B.-S. Yang, J. Lee, S. Lee, S. Park, Y. C. Chung, S. Kim, K. Ebcioglu, E. Altman, and S.-M. Moon. Efficient register mapping and allocation in latte, an open-source java just-in-time compiler. *IEEE Trans. Parallel Distrib. Syst.*, 18:57–69, January 2007.

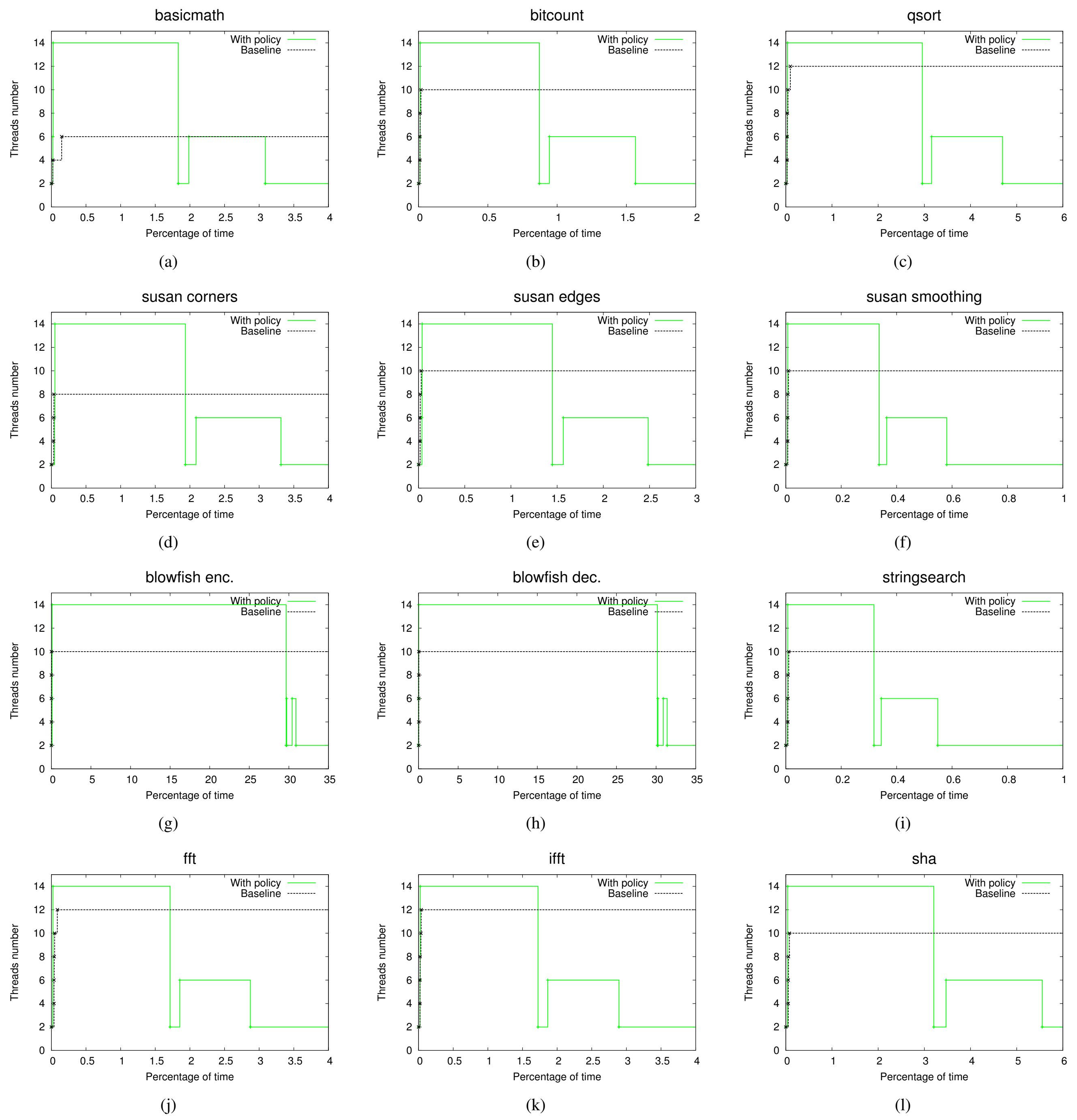

Fig. 9. Static threads used with and without the policy described in Section III-C2.