

Pervasive Parallelism for Managed Runtimes

Albert Noll
ETH Zurich, Switzerland
albert.noll@inf.ethz.ch

Thomas R. Gross
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

Modern programming languages like C# or Java are executed in a managed runtime and offer support for concurrency at a high level of abstraction. However, high-level parallel abstractions (e.g., thread pools) can merely be provided as a library since the underlying runtime (including the dynamic compiler) is aware of only a small set of low-level parallel abstractions.

In this paper we discuss alternative abstractions of concurrency in the source language, the runtime, and the dynamic compiler. The abstractions enable a dynamic optimizing compiler to perform new code transformations that can adapt the granularity of parallel tasks according to the system resources. The presented optimizations allow the runtime to tune the execution of parallel code fully automatically.

1 Introduction

Modern programming languages like C# or Java are executed in a managed runtime and offer support for concurrency at a high level of abstraction. However, since high-level parallel abstractions (e.g., thread pools) are provided as a library, the managed runtime and the dynamic compiler are unaware of these abstractions. Although managed runtimes can gather profile information [2] about the running application, the execution of parallel code is determined by the implementation of high-level parallel constructs in the API that does not have such profiling information.

There are several factors that influence the performance of parallel code. First, fine-grained parallel task creation and termination incurs an overhead that seriously degrades the execution time of parallel code [19]. If, however, the task size is too coarse-grained, load balancing can be not optimal. The optimal task size depends on the target system (i.e., the number of cores) and cannot be determined statically.

Second, resource contention can add overhead to the execution time of parallel code. There exist several approaches to mitigate these effects. E.g., Zhuravlev et al. [21] use a contention-aware scheduler to use shared resources more efficiently. Finally, synchronization constructs can be a performance bottleneck of parallel code. Feedback-driven threading is a technique that uses data provided by hardware performance counters to find the optimal number of threads in an OpenMP program [16].

This paper discusses three high-level parallel abstractions that are added to the source language, the runtime and the dynamic compiler. The abstractions provide the runtime and dynamic compiler with significantly more opportunities to optimize the execution of parallel code. The abstractions do not require modifications the source language and allow a programmer to incrementally parallelize existing applications. The remainder of this paper discusses the abstractions in the context of the Java platform, since Java has a well-defined memory model [12].

The first abstraction is a *parallel task*, and is added to the source language, the Java Virtual Machine (JVM) [11] and the dynamic compiler. A parallel task is a unit of work that can be executed by a separate thread. In the source language, we abstract parallel tasks as parallel calls [5, 17] or parallel closures rather than objects. One advantage is that no task objects must be allocated, initialized, and garbage collected. Such a design reduces the parallel task creation and termination overhead. In Java, a parallel call can be provided by an annotation and a JVM that implements the semantics of the annotation. The advantages and drawbacks of a parallel calls/closures are discussed in Section 3.1. The representation of a parallel call in the JVM and the dynamic compiler is discussed in Section 3.2.

The second high-level parallel abstraction is a *thread pool*. The JVM is extended by an internal thread pool that executes parallel tasks. An internal thread pool provides the JVM with the opportunity to (1) collect profile information of parallel tasks and (2) determine an execu-

```

1 final int x = ...;
2 for (int i = 0; i < N; i++) {
3     threadPool.execute(new Runnable() {
4         public void run() {
5             int x1 = x + 1;
6             doSomething(x1);
7         }
8     });
9 }

```

Figure 1: Motivating example.

tion order that maximizes resource utilization [21].

Finally, the synchronization construct *sync* is provided by a method (`sync()`) that is added to the type `Object` and has (similar to `wait()`) a special meaning to the JVM and the dynamic compiler. `sync()` is used to synchronize the execution of parallel tasks (Section 3.1).

The dynamic compiler explicitly represents a parallel task and the synchronization of parallel tasks in the intermediate representation (IR). Based on the explicit representation, we present two scheduling optimizations that enable the JVM to increase or decrease the parallel task granularity. Furthermore, we present new code optimizations that are not available to traditional dynamic compilers.

2 Motivating example

Figure 1 shows an example in which parallel task objects are submitted to a thread pool. The parallel tasks are independent and can be processed in an arbitrary order. Traditional dynamic compilers have little potential for optimization since the task objects escape [9] the current thread. Consequently, optimizations such as stack allocation or synchronization removal [4] cannot be applied.

However, the code in Figure 1 can be optimized by merging two or more tasks into a single task (task merging). Such a transformation reduces the overhead of task creation and scheduling and is particularly effective if the task size is small. Traditional dynamic compilers cannot perform task merging, since the notion of a parallel task (e.g., `Runnable`) is not known to the Java Virtual Machine (JVM) [11].

Furthermore, the computation of `x1` in line 5 of Figure 1 is loop invariant and can be hoisted above the loop. This code transformation saves N additions. Hoisting the computation of `x1` out of the task object is not possible in traditional dynamic compilers, since the `run()` method is called by a thread in the thread pool and the defining context of the task object (lines 1–3 and line 9 in Figure 1) is not considered for code optimizations. The rest of the paper describes extensions to the source lan-

```

1 void foo(int x) {
2     parallelCall(1);
3     parallelCall(2);
4     sync();
5 }
6 @Parallel
7 void parallelCall(int x) {
8     System.out.println(x);
9 }

```

Figure 2: Parallel call example.

guage, JVM and the dynamic compiler that allow such code transformations.

3 Implications for the Java platform

The optimizations that are described in Section 2 are enabled by extensions to the source language (Section 3.1), the JVM, and the intermediate representation (ParIR) of the dynamic compiler (Section 3.2).

3.1 Source-language abstractions

Java uses object-orientation to abstract parallelism. E.g., a task can be represented as an object that implements the `Runnable` interface. A task object is attached to a thread for execution. Consequently, the execution of parallel code is always associated with the overhead from object creation and the resulting garbage collection. A recent study has shown that object allocation sacrifices the performance of parallel code [20].

The JVM and the dynamic compiler are also unaware of a parallel tasks or thread pools. As a result, the JVM cannot determine an execution order of independent parallel tasks that maximizes shared system resource utilization. However, a JVM that is aware of parallel tasks and has one or more internal thread pools can use profile information (e.g., gathered by hardware performance counters) to maximize resource utilization through task scheduling (similar to [21], but at the JVM level).

To make the JVM aware of tasks we identify two source language abstractions: parallel calls and parallel closures. A parallel call is a method that can be executed by a separate thread (callee thread), and the caller need not wait for the callee to return. Figure 2 shows an example of two consecutive parallel calls. The function `parallelCall()` is annotated with `@Parallel`. The execution of function `foo()` in Figure 2 can either print "1 2" or "2 1", depending on the scheduling.

A parallel closure is a closure that is potentially executed in a separate thread. Similar to a parallel call, a parallel closure can be annotated with `@Parallel`. Both, a parallel call and a parallel closure can be mapped

```

1 pStart
2 invokeVirtual //parallelCall;
3 pEnd
4 pStart
5 invokeVirtual //parallelCall;
6 pEnd
7 pSync;

```

Figure 3: Parallel IR of method `foo()`.

to ParIR (see Section 3.2). The rest of this section discusses the design of a parallel calls, since it is still uncertain if closures will be included in Java 8. Parallel calls, however, can be used to incrementally parallelize existing Java applications.

The built-in method `sync()` causes the caller thread (thread that executes `foo()`) to block until every callee thread (including recursively spawned parallel calls) returns from the parallel call.

A parallel call can return a value. However, the return value is guaranteed to be set correctly only after calling the `sync()` method. A call to `sync()` also guarantees that all changes to global memory of the callee thread are available to the caller thread. The implications of a parallel call on the Java Memory Model (JMM) [12] are discussed in Section 3.2. Parallel calls are not bound to a specific type and can have an arbitrary signature. E.g., `static` and `private` methods can be parallel calls.

The integration of a parallel call into the JVM provides several dynamic optimization opportunities. E.g., the JVM can decide at runtime if a parallel call is executed by the caller thread or by a separate callee thread. Such a decision can be based on the availability of resources or the (estimated) task size. In traditional Java, the programmer must explicitly describe under which conditions a task is executed in parallel. The decision made by the programmer cannot be overridden by the JVM. The reason is that the JVM does not know the semantics of a task or a thread pool. From the JVM's point of view, a reference to one object (task) is passed to another object (thread pool). Our approach provides the JVM with the opportunity to dynamically optimize for the most efficient level of parallelism.

3.2 Parallelism-aware IR

Figure 3 shows the parallelism-aware intermediate representation (ParIR) of method `foo()`. The translation from byte code to ParIR expands a parallel call to make parallelism explicit in the IR. Explicit parallelism is provided through three new IR nodes: `pStart` marks the beginning of a code region that is potentially executed by a separate thread. Such a code region is called parallel region. The call to `parallelCall()` is a regular

method call. `pEnd` marks the end of the parallel region. Finally, `pSync` represents the semantics of the `sync()` function.

A parallel region can contain arbitrary instructions. E.g., the dynamic compiler can inline the body of `parallelCall()` into the body of `foo()`. The inlined code can still be executed in a separate thread. More details about parallel call inlining can be found in Section 4.2.1.

The new IR nodes have implications on the JMM since the callee thread must see all updates to global memory that happened-before `pStart`. Similarly, the caller thread must see all writes to global memory that are performed by the callee thread upon returning from the `sync()` method. In terms of the JMM, `pStart` has release semantics (the caller thread must commit the local view of memory to global memory), and `pSync` has acquire semantics (all modifications to global memory of the callee thread are visible to the caller thread).

Furthermore, the compiler is not allowed to arbitrarily re-order instructions around a parallel region. In general, `pStart` and `pEnd` must not be reordered with any instruction. If, however, the dynamic compiler can prove that an instructions is parallelism-invariant (Section 4.2.2), the compiler can re-order the instruction with `pStart` or `pEnd`.

One advantage of ParIR over a traditional IR is that the dynamic compiler can generate a sequential version of a parallel call. A sequential call can be beneficial, if the program is executed on a single-core, or profiling shows that a parallel call, if executed by a separate thread, results in a performance loss. The sequential performance of the parallel call is not sacrificed since parallelism can be completely eliminated by the dynamic compiler. A sequential execution of parallel code that uses the traditional Java threading API is likely to incur a performance penalty due to object creation and task scheduling.

4 Compiler optimizations

This section presents dynamic compiler optimizations that are enabled by parallel calls in combination with the parallelism-aware IR. Section 4.1 presents two code transformations that can decrease the granularity of parallelism. Section 4.2 presents two code optimization techniques: parallel call inlining and parallelism-invariant code motion.

4.1 Scheduling optimizations

Scheduling optimizations aim at adapting the granularity of parallel tasks according the underlying system configuration. A fine task granularity has the advantage of good load balancing, but incurs larger scheduling overhead. A

```

1  for(int i = 0; i < 16; i++) {
2    pStart
3    invokeVirtual //parallelCall;
4    pEnd
5  }
6  pSync

```

(a) Original code.

```

1  for (x = 0; x < 16; x+=4) {
2    pStart
3    for (i = x; i < x + 4; i++) {
4      invokeVirtual //parallelCall;
5    }
6    pEnd
7  }
8  pSync;

```

(b) After optimization.

Figure 4: Task merging in the presence of a loop.

coarse task granularity offers inferior load balancing, but incurs less scheduling overhead.

4.1.1 Parallel call merging

One way to reduce the scheduling overhead is to decrease the number of parallel calls. A parallel call can be eliminated by either converting the parallel call to a sequential call or by merging two parallel calls. E.g., the two parallel calls in Figure 3 can be merged into a single parallel region. Parallel call merging is a legal code transformation, since the merging of two consecutive parallel regions simply specifies one legal execution order of the two parallel calls: The first call to `parallelCall()` always happens before the second call to `parallelCall()`, since the calls are executed by the same thread.

Merging two parallel calls increases the parallel task granularity. Parallel call merging is beneficial if the scheduling overhead is larger than the execution time of the parallel call. Traditional dynamic compilers cannot perform such a code transformation since the notion of a task is not part of the JVM specification.

4.1.2 Loop invariant parallel call merging

Figure 4 presents an example of loop invariant parallel call merging. Figure 4(a) shows the ParIR of the original program code. The caller thread performs 16 parallel calls (lines 1–5), and waits for all spawned parallel calls to finish (line 6). The original program code implicitly contains the information that the parallel calls can be processed in an arbitrary order. In other words, the execution order of the parallel calls is loop invariant.

Similar to parallel call merging, loop invariant parallel

call merging determines a specific execution order of a set of parallel calls to reduce the schedule overhead. Figure 4(b) shows how the dynamic compiler reduces the scheduling overhead. Let us assume that the code runs on a four cores. Consequently, the compiler encloses the parallel region (line 2–6) with a loop that has four iterations (lines 1–7) and adapts the loop stride and bounds accordingly. This optimization reduces the scheduling overhead by a factor of four. The reduced scheduling overhead is at the expense of work balancing. Loop invariant parallel call merging is well suited for regular computations in which the execution time of each loop iteration is approximately the same.

Parallel call merging is a good candidate for online feedback-directed optimization [2]. If the profiling system detects that the execution time of a parallel call is small the JVM can decide to recompile and merge parallel calls.

4.2 Code optimizations

In contrast to scheduling optimizations, code optimizations aim at improving the code quality by applying compiler optimizations in the context of parallel regions.

4.2.1 Parallel call inlining

Method inlining is an optimization that has significant impact on the system performance [15]. Inlining expands the target method by the body of the callee(s) and therefore extends the scope of the compiler. Due to the representation of a parallel call in ParIR, a parallel call can be inlined like a sequential method call. However, the compiler must make sure that all instructions that are inlined from the parallel call are contained in the parallel region. If a parallel call is inlined, the stack frame of the callee thread must contain the same values as the stack frame of the caller thread. The compiler and the runtime system take care of that initialization. The code of the inlined parallel call can profit from standard optimizations such as copy/constant propagation or common sub-expression elimination.

4.2.2 Parallelism-invariant code motion

Parallelism invariant code motion is similar to loop invariant code motion. Code that is executed more than once (in a loop or a parallel region), always gives the same results and is side-effect free can be replaced by the computational result.

Figure 5 provides an example of a combination of parallel call inlining, parallelism invariant and loop invariant code motion. The original code is given in Figure 5(a) and the optimized version in Figure 5(b). In the original

```

1 void foo(int x) {
2   for (int i = 0; i < N; i++) {
3     pStart
4     invokeVirtual //parallelCall;
5     pEnd
6   }
7 }
8 @Parallel
9 void parallelCall(int x) {
10  x = x + 1;
11  doSth(x);
12 }

```

(a) Original code.

```

1 void foo(int x) {
2   x1 = x + 1;
3   for (int i = 0; i < N; i++) {
4     pStart
5     invokeVirtual //doSth;
6     pEnd
7   }
8 }

```

(b) After optimization.

Figure 5: Parallelism-invariant code motion.

code, the programmer forks a set of independent parallel calls (lines 2–6). Each parallel call takes one parameter that is incremented and passed to another function. Incrementing x in the parallel call is parallelism invariant, since there is no assignment to x in function $foo()$ (i.e., x is final).

In the optimized version, the parallel call is first inlined into method $foo()$. In the next step, parallelism invariant code motion detects that the increment of variable x can be moved out of the parallel region. This optimization is a legal code transformation, since x is a local variable and changes to local variables cannot be detected by other threads. Since Java passes arguments call-by-value, a new local variable $x1$ is generated. Finally, loop invariant code motion hoists the computation of $x1$ out of the loop. The optimized version in Figure 5 saves one addition per parallel call.

5 Related work

There exists a large body of related work in the area of compiler optimizations for parallel programs. Lee et al. [10] present compiler algorithms for explicit parallel programs that enforce sequential consistency. Other examples include a reaching definition analysis [6], a bit-vector analysis [8], or the definition of a parallel program graph [14]. The related work in the area of compiler analysis for parallel programs is a good foundation to implement the optimizations that are discussed in the paper.

Cilk [5] extends the C/C++ programming language by

keywords that spawn a parallel call (`spawn`) or synchronize the spawned parallel calls (`sync`). The semantic of a parallel call corresponds to the semantics as proposed in Cilk. The main difference to Cilk is that our approach integrates a parallel call into a runtime system that allows dynamic optimizations such as profile-based recompilation [18]. The current Cilk runtime cannot perform such optimizations.

X10 [3] provides explicit parallelism through keywords. In [19] the authors present a compiler framework that aims at reducing the task-creation and termination overhead. The optimizations are similar to parallel call merging and loop invariant parallel call merging. However, the authors use a static compilation approach, which makes the underlying runtime oblivious of the parallel constructs. Consequently, adaptive optimization techniques (e.g., adaptive parallel call merging) cannot be applied.

The Erlang [1] language and runtime system is designed for parallel computation. Processes only communicate via message passing. Unlike in most JVMs the runtime system creates and schedules processes. The Erlang runtime has an internal thread pool to execute processes.

Most approaches to describe parallelism are library-based. E.g., Intervals [13] provide an abstraction to explicitly specify the execution order of tasks. Parallel Java [7] provides OpenMP and MPI constructs for Java. In both approaches, the JVM and the dynamic compiler is unaware of the parallel constructs.

6 Conclusion

Multi-core computers are de facto standard. However, managed runtimes and dynamic compilers have not yet followed the shift from sequential to parallel computers. The support for parallelism is limited to a small number of low-level primitives, and high-level parallel constructs are provided as a library. As a result, dynamic compilers perform only optimizations that are effective for sequential code. Optimizations that are particularly targeted at parallel code are not supported in state-of-the-art dynamic compilers.

This paper aims at increasing the knowledge of a managed runtime and the dynamic compiler about parallel constructs. The additional knowledge enables the dynamic compiler to perform new optimizations. The optimizations described in this paper are good candidates for online feedback-directed optimizations, since the granularity of parallel tasks can be adapted at runtime.

Writing correct parallel programs is a hard task. Tuning the performance of parallel programs should be performed fully automatically by the runtime system. Our approach shows one solution to this problem.

7 Acknowledgements

I want to thank Mathias Payer and Zoltan Majo for their valuable discussions and helpful comments.

References

- [1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- [2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *OOPSLA '02*, pages 111–129, NY, USA, 2002. ACM.
- [3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.
- [4] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25:876–910, November 2003.
- [5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.
- [6] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. *SIGPLAN Not.*, 28:159–168, July 1993.
- [7] A. Kaminsky. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *IPDPS*, pages 1–8, 2007.
- [8] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 18:268–299, May 1996.
- [9] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE '05*, pages 111–120. ACM, 2005.
- [10] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *PPoPP '99*, pages 1–12, NY, USA, 1999. ACM.
- [11] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [12] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 378–391. ACM, 2005.
- [13] N. D. Matsakis and T. R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA '10*, pages 634–651, NY, USA, 2010. ACM.
- [14] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC '97*, pages 94–113, London, UK, 1998. Springer-Verlag.
- [15] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a Java just-in-time compiler. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*, pages 91–104. USENIX Association, 2002.
- [16] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *ASPLOS XIII*, pages 277–286, NY, USA, 2008. ACM.
- [17] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *PPoPP '99*, pages 60–71, NY, USA, 1999. ACM.
- [18] J. Whaley. Partial method compilation using dynamic profile information. *SIGPLAN Not.*, 36:166–179, October 2001.
- [19] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *PACT '10*, pages 169–180, NY, USA, 2010. ACM.
- [20] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *OOPSLA '09*, pages 361–376, NY, USA, 2009. ACM.
- [21] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10*, pages 129–142, NY, USA, 2010. ACM.