Online Feedback-Directed Optimizations for Parallel Java Code

Albert Noll
Department of Computer Science
ETH Zurich

Thomas Gross
Department of Computer Science
ETH Zurich

Abstract
The performance of parallel code significantly depends on the parallel task granularity (PTG). If the PTG is too coarse, performance suffers due to load imbalance; if the PTG is too fine, performance suffers from the overhead that is induced by parallel task creation and scheduling.

This paper presents a software platform that automatically determines the PTG at run-time. Automatic PTG selection is enabled by concurrent calls, which are special source language constructs that provide a late decision (at run-time) of whether concurrent calls are executed sequentially or concurrently (as a parallel task). Furthermore, the execution semantics of concurrent calls permits the runtime system to merge two (or more) concurrent calls thereby coarsening the PTG. We present an integration of concurrent calls into the Java programming language, the Java Memory Model, and show how the Java Virtual Machine can adapt the PTG based on dynamic profiling. The performance evaluation shows that our runtime system performs competitively to Java programs for which the PTG is tuned manually. Compared to an unfortunate choice of the PTG, this approach performs up to 3x faster than standard Java code.

Categories and Subject Descriptors D.3.4 Processors [Programming Languages]: Compilers

General Terms Languages, Performance

Keywords JIT compilation, dynamic optimization, feedback-directed optimization

1. Introduction
Writing fast and scalable parallel code is difficult. The reason is that the performance of parallel code depends on numerous hardware and software properties that are not always known (and obvious) to the programmer. One such important property is the parallel task granularity (PTG). The PTG describes the tradeoff between the overhead that is associated with parallel task execution (e.g., object allocation, scheduling) and the potential performance gain. A fine PTG provides better load balance than a coarse PTG. However, executing fine-grained parallel tasks incurs more run-time overhead than executing coarse-grained parallel tasks. Figure 1 illustrates this tradeoff.

![Figure 1. Performance impact of parallel task granularity.](image_url)

The x-axis in Figure 1 shows the number of parallel tasks used to execute a fixed amount of work (W). For example, a value of 2 on the x-axis corresponds to an execution in which W is split into 2 equally-sized parallel tasks. The y-axis shows the speedup over a sequential execution on a 32-core system.

The graph in Figure 1 can be divided into three regions. The left region is characterized by a poor load balance and a low overhead. The load is imbalanced, since a 32-core system requires at least 32 parallel tasks to provide all cores with work. The overhead is low, since only a small number of parallel tasks is generated. The middle region has a good load balance (a large number of parallel tasks) but incurs a significant overhead due to parallelism. For 100,000 parallel tasks or more, the parallel version performs slower than the sequential version. To summarize, parallelism is underspecified in the left region, well-
specified in the middle region, and overspecified in the right region. Both, overspecification as well as underspecification of parallelism have a significant negative impact on the performance.

Unfortunately, current programming techniques support only a manual determination of the PTG. In particular, programmers must decide which parts of an application are best executed in parallel and what PTG yields good performance. At the same time, we see a wide range of parallel systems in use, ranging from modest dual-core systems to systems with 64 cores (or more). Ahead-of-time custom-tuning software for each platform is expensive, if possible at all.

This paper presents a software platform that automatically determines the PTG at run-time. Our system asks the programmer to overspecify parallelism. The runtime system evaluates the PTG as provided in the source code and adapts the PTG towards a good tradeoff between load balance and parallelism overhead. More specifically, we show that the runtime system effectively removes overspecified parallelism by (i) serializing parallel tasks that incur a high overhead and (ii) merging two (or more) parallel tasks into a single parallel task. However, if performance suffers from parallelism underspecification, the runtime system must auto-parallelize (parts of) the application to speed up execution. Since current runtime systems cannot effectively auto-parallelize applications, performance problems due to parallelism underspecification cannot be fixed by the run-time system.

The automatic determination of the PTG is enabled by concurrent calls. Concurrent calls are special source language constructs that can be executed sequentially or in parallel. We discuss the integration of concurrent calls into the Java programming language and the Java Virtual Machine (JVM), including the just-in-time compiler (JIT compiler), and show how the JVM adapts the PTG. In particular, we present a compiler analysis that determines if concurrent calls can be merged (collapsed into a single concurrent call) and the merged concurrent call provides sequential consistency [12] for data-race free programs. We therefore present an integration of concurrent calls into the Java Memory Model (JMM) [16]. The performance evaluation of a prototype implementation in the Jikes Research Virtual Machine (Jikes RVM) [2] shows that our runtime system performs competitively to Java programs for which the PTG is tuned manually. Compared to an unfortunate or unlucky choice of the PTG, our approach performs up to 3X faster than standard Java code.

2. System design

This section presents an overview of how concurrent calls are integrated into the Java platform. We chose the Java platform for the following reasons: First, Java has a well-defined memory model [16] that exactly defines the semantics of parallel Java code. Second, Java code runs in a managed run-time environment, the Java Virtual Machine (JVM), which is typically equipped with a JIT compiler and a dynamic profiling infrastructure. Third, Java supports annotations, which can be used to attribute Java language constructs without having to change the language itself. Finally, there exist several open source implementations of the JVM.

| Source language extensions | @Concurrent @Sync |
| Runtime system extensions | JIT compiler Profiling system Adaptive optimization system |

Figure 2. System overview.

Figure 2 depicts the system overview. We extend the Java programming language by two annotations: @Concurrent and @Sync. @Concurrent provides the declaration of concurrent calls. @Sync provides the declaration of synchronization calls, which can be used to synchronize concurrent calls.

The runtime system extensions include extensions to the just-in-time compiler (JIT compiler), the profiling system, and the Adaptive Optimization System (AOS), which identifies methods that are recompiled by the JIT compiler. We extend the JIT compiler by a new intermediate representation (IR) that represents concurrent calls and synchronization calls explicitly. This IR is called concurrency-aware intermediate representation (CIR). The JIT compiler optimizes the parallel task granularity (PTG) in CIR.

Standard profiling systems of modern JVMs are designed to track down performance bottlenecks of sequential code. We present the design of a concurrency-aware dynamic profiler that measures the behavior of parallel code and provides the necessary information for deciding which concurrent calls are compiled to sequential code, to parallel code, and what concurrent calls are best merged to achieve good performance.

The AOS of a JVM selects methods that are good candidates for dynamic recompilation. We extend the standard AOS by the parallel code execution optimization (PCEO) algorithm, which uses profile information collected by the extended profiler to analyze the behavior of a particular PTG. The PCEO algorithm can change a particular PTG by triggering a recompilation that, e.g., merges concurrent regions that were not merged in the previous configuration.

3. Java semantics of @Concurrent and @Sync

A method declaration that includes the @Concurrent annotation declares a concurrent method. Concurrent methods can be executed sequentially or in parallel. The semantics of a sequential execution and a parallel execution of a concurrent method are defined as follows:
• The sequential execution of a concurrent method results in the same execution as if the method is not annotated.

• The parallel execution of a concurrent method has the same semantics as if a separate thread executes the method call. I.e., a concurrent method executes asynchronously with instructions that follow the concurrent method. The separate thread that executes a concurrent call is called child thread and the thread that encounters the concurrent method is called parent thread.

Sequential – as well as the parallel – executions of concurrent methods must result in legal program behavior. It is the responsibility of the programmer to identify code regions that have such a behavior. Other explicit parallel programming languages have similar requirements. For example, the Cilk programming language [6] has the same requirements for the spawn keyword as our approach has for concurrent calls. Another popular example that has similar requirements is OpenMP [1]. Parallel regions in OpenMP are executed by a team of N threads. The team consists of the master thread and N - 1 child threads. The master thread is the thread that encounters the parallel region. If the OpenMP runtime system decides to use only a single thread to execute a parallel region that thread can be the master thread. This setup corresponds to a sequential execution of a parallel region.

A method that is annotated with @Sync is called synchronization method. Synchronization methods can be used to synchronize concurrent calls. In particular, the execution of a synchronization method suspends the calling thread prior to executing the method until all concurrent methods that are issued by the calling thread are retired. A concurrent method retires if all effects of the concurrent method are visible to the calling thread. A synchronization method proceeds immediately if there are no concurrent methods in flight.

Figure 3 shows how concurrent methods are declared, called, and synchronized. Figure 3(a) shows the declaration of the concurrent method C() and the synchronization method S(). If C() is called with the parameter x = 1 the output of the program shown in Figure 3(b) can either be "1 2" or "2 1". Since there is no synchronization between the concurrent method calls in line 2 and line 3, and concurrent methods execute asynchronously (if executed in parallel), the order in which the concurrent methods are executed is undefined.

The two concurrent calls return a value. The variable that stores the return value must be declared final. The return value is initialized to the default value of the specified type (e.g., null for a reference type) and contains the return value only after the execution a synchronization method. The return values in Figure 3(b) are accessed correctly, since the synchronization method is called before the return values are accessed. Consequently, the return value of foo() is always 5. The JVM initializes the parameters of concurrent calls and sets the return values.

3.1 Java language integration

This section discusses the Java language integration of concurrent calls. More specifically, we present concurrent calls in the context of polymorphism, exceptions, and discuss the interaction with the standard Java threading API.

3.1.1 Polymorphism

The current design supports the @Concurrent and the @Sync annotation for private (and private static) methods. In Java, private methods can be called only from the class in which the method is declared. As a result, a concurrent method cannot be called “accidentally” by e.g., deriving from a class and calling the concurrent method of the superclass. The current design is conservative, since it allows only the implementor of a class to invoke concurrent calls. We choose such a design, since the programmer must ensure that concurrent calls yield a legal program execution. However, the current design can be changed easily and changing the design (i.e., relaxing the constraints which methods can be annotated with @Concurrent and @Sync) has no effect on the runtime system extensions.

3.1.2 Exception handling

The execution of concurrent methods and synchronization methods can raise unchecked exceptions. An uncaught exception aborts the execution of the concurrent method. If the exception occurs in the child thread, the JVM propagates the exception to the parent thread. Exceptions that occur in a synchronization method are handled according to the Java language specification.

3.1.3 Interaction with the Java threading API

Concurrent methods and synchronization methods are alternative constructs to describe concurrency in the Java source language and can be used in combination with the standard library-based Java threading API. Child threads are regular Java threads that can acquire and release monitors, wait on objects, and notify (be notified by) other threads. One difference of declaring concurrency using concurrent calls compared to using standard Java threads is that concurrent methods add concurrency to existing applications without having to define new classes and to instantiate new objects. As a result, the sequential execution of concurrent methods is po-

![Figure 3](image-url)
tentially faster than the sequential execution of parallel Java code.

4. JIT compiler extensions

Concurrent methods and synchronization methods are represented explicitly in the intermediate representation (IR) of the JIT compiler. The standard, sequential IR is extended by four new IR nodes that describe concurrent regions and the synchronization of concurrent regions. An IR that contains at least one of the four new IR nodes is called CIR. The four new IR nodes are defined as follows:

- cStart: marks the beginning of a concurrent region. Instructions that follow the cStart CIR-node are part of the same concurrent region. All instructions in a concurrent region are executed by the same thread (parent thread or child thread).
- cEnd: denotes the end of a concurrent region.
- cBarrier: represents a boundary across which the JIT compiler is not allowed to reorder instructions. I.e., the code generated by the JIT compiler must ensure that all instructions that are ordered before a cBarrier CIR-node are executed and retired before all instructions that follow the cBarrier CIR-node. The cBarrier CIR-nodes guarantees sequential consistency [12] for data-race free programs, as explained in Section 4.1.
- cSync: blocks until all concurrent methods that are issued by the thread that encounters cSync have retired.

The example in Figure 4(a) shows the implementation of method `foo()`, which contains two concurrent methods and one synchronization method. Figure 4(b) shows the representation of `foo()` in the CIR. The method declarations of `C()` and `S()` are taken from Figure 3(a).

```
1  int foo(int x) {
2      final int r1 = C(x);
3      final int r2 = C(x+1);
4      S();
5      return r1 + r2;
6  }
```

(a) Example with two concurrent methods and one synchronization method.

```
1  cStart;  cBarrier;  
2  r1 = call C(x);  cBarrier;  cEnd;  
3  r2 = call C(t1);  cBarrier;  cEnd;  
4  cSync;  cBarrier;  
5  call S();  
6  cBarrier;  
7  return r1 + r2;  
```

(b) CIR of Figure 4(a).

**Figure 4.** Representation of concurrent methods and synchronization methods in the source code and the CIR.

All instructions of a concurrent region are enclosed between two `cBarrier` CIR-nodes, a `cStart` and a `cEnd` CIR-node. The `cSync` CIR-node is followed by a `cBarrier` CIR-node, the synchronization method call, and another `cBarrier` CIR-node. The JIT compiler compiles concurrent methods and synchronization method calls to regular IR nodes.

4.1 Integration of CIR into the JMM

The Java Memory Model (JMM) [16] defines the semantics of parallel Java code. The JMM is a relaxed-consistency model that allows the JIT compiler (and the hardware) to reorder independent instructions. Despite the possible instruction reordering, the JMM guarantees sequential consistency (SC) [12] for correctly synchronized programs, i.e., programs that contain no data races. The JMM defines strict rules that define precisely which values can be obtained by a read of a shared field that is updated by multiple threads. Concurrent methods and synchronization methods can read from and write to shared memory. The explicit representation in the CIR requires a precise definition of the JMM semantics to ensure compliance with the JMM.

**JMM semantics of cStart and cEnd**: Concurrent regions begin with the `cStart` CIR-node. `cStart` is an inter-thread action (an action that can be observed by another thread), since the instructions of a concurrent region can be executed in parallel with instructions that follow the concurrent region in program order. As such, `cStart` semantically starts a new thread that executes the concurrent region. Since starting a thread is an inter-thread action in the JMM, the `cStart` CIR-node is an inter-thread action as well. Starting a new thread has release semantics in the JMM. Release semantics imply that all inter-thread actions that happen before `cStart` must be visible to and ordered before all inter-thread actions after `cStart`. To provide release semantics, the local view of memory (e.g., field values that are cached in registers) of the parent thread is transferred to the child threads. `cStart` has acquire semantics for the child threads, because child threads must update their local view of memory with global memory.

The `cEnd` CIR-node delimits the end of a concurrent region. `cEnd` represents the final inter-thread action of a concurrent region. Consequently, all inter-thread actions in the concurrent region must be ordered before `cEnd` and all inter-thread actions that follow `cEnd` in program order must happen after `cEnd`. Child threads commit the local view of memory to global memory. Consequently, `cEnd` has release semantics for child threads.

**JMM semantics of cBarrier**: The `cBarrier` CIR-node represents a barrier across which the JIT compiler is not allowed to reorder instructions. As discussed in Section 4.4, two (or more) concurrent regions can be merged into one concurrent region. Merged concurrent regions are seen by the JIT compiler as a single concurrent region and the reordering rules for intra-region reorderings (Section 6) ap-

---

2 We use the term instruction (or IR node) in the JVM context.
ply. However, reordering the instructions of merged concurrent regions can introduce new behaviors. Consequently, the cBarrier node ensures that concurrent region merging remains a valid code transformation in the JMM. Consider the following example, which is taken from [16]:

Initially, \( x == y == 0 \)

<table>
<thead>
<tr>
<th>foo()</th>
<th>C1()</th>
<th>C2()</th>
<th>C3()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: C1();</td>
<td>1: r1=x;</td>
<td>4: r2=x;</td>
<td>6: r3=y;</td>
</tr>
<tr>
<td>2: C2();</td>
<td>2: if(r1==0)</td>
<td>5: y=r2;</td>
<td>7: x=r3;</td>
</tr>
<tr>
<td>3: C3();</td>
<td>3: x=1;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Caller. (b) Callees.

Figure 5. \( r1 == r2 == r3 == 1 \) is not legal.

Figure 5(a) contains three concurrent calls. The method bodies of the three concurrent calls C1(), C2(), and C3() are illustrated in Figure 5(b). x and y are normal fields that can be accessed by three different threads. r1, r2, and r3 are local registers. An execution in which \( r1==r2==r3==1 \) is not legal in the JMM. The reason is that \( r1 \) must be 0 provided that 1 is assigned to \( x \). The behavior in question \( (r1==r2==r3==1) \) requires "out-of-thin-air" reads, which violate the security guarantees of Java.

Figure 6 shows the effects of a compiler optimization that results in an out-of-thin-air read that is possible without the cBarrier CIR-node. More precisely, the code in Figure 6 illustrates a variant of the code shown in Figure 5 that allows \( r1==r2==r3==1 \).

Initially, \( x == y == 0 \)

<table>
<thead>
<tr>
<th>foo()</th>
<th>C1() and C2()</th>
<th>C3()</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: cStart;</td>
<td>10: r3=y;</td>
<td>10: r3=y;</td>
</tr>
<tr>
<td>1: cBarrier;</td>
<td>11: x=r3;</td>
<td>11: x=r3;</td>
</tr>
<tr>
<td>2: r1=x;</td>
<td>2: y=1;</td>
<td></td>
</tr>
<tr>
<td>3: if(r1==0)</td>
<td>3: r1=x;</td>
<td></td>
</tr>
<tr>
<td>4: x=1;</td>
<td>4: if(r1==0)</td>
<td></td>
</tr>
<tr>
<td>5: cBarrier</td>
<td>5: x=1;</td>
<td></td>
</tr>
<tr>
<td>6: r2=x;</td>
<td>6: cBarrier</td>
<td></td>
</tr>
<tr>
<td>7: y=r2;</td>
<td>7: r2=1;</td>
<td></td>
</tr>
<tr>
<td>8: cBarrier;</td>
<td>8: cBarrier;</td>
<td></td>
</tr>
<tr>
<td>9: cEnd;</td>
<td>9: cEnd;</td>
<td></td>
</tr>
</tbody>
</table>

(a) Effects of "thread inlining". (b) Code optimizations across concurrent regions.

Figure 6. Must not allow: \( r1==r2==r3==1 \).

Figure 6(a) shows the code in which the JIT compiler inlined the calls C1() and C2() into function foo(). In the inlined version, the JIT compiler can determine that the only legal values for \( x \) and \( y \) are 0 and 1. Consequently, the JIT compiler can replace the read of \( x \) in line 6 by 1 and line 7 by \( y = 1 \). If the JIT compiler reorders instructions across cBarrier nodes, as shown in Figure 6(b), the behavior in question \( (r1==r2==r3) \) is possible. The following execution order yields the behavior in question: 0, 1, 2, 10, 11, 3-9. The cBarrier CIR-node prohibits this illegal behavior, since instruction reorderings are restricted to happen within a concurrent region, i.e., the body of a concurrent call.

JMM semantics of cSync: The cSync CIR-node is an inter-thread action since the parent thread can determine if all child threads have finished execution. In terms of the JMM, cSync has the same semantics as calling join() on every child thread. The join() operation ensures that all children have finished execution and updated their local view of memory to global memory. The parent thread synchronizes the local memory with global memory.

4.2 Using existing compiler optimizations with CIR

This section discusses the interoperability of the CIR with existing compiler transformations. Existing code transformations can reorder independent intra-thread actions as well as independent inter-thread actions. Reordering optimizations in CIR can be classified into intra-region and inter-region reordering optimizations. An intra-region reordering optimization reorders an IR-node \( N_0 \) with a set of target IR-nodes \( N_T = \{N_1, N_2, ..., N_n\} \) and \( N_T \) does not contain a CIR-node. Similarly, an inter-region reordering optimization reorders an IR node \( N_0 \) with a set of target IR-nodes \( N_T = \{N_1, N_2, ..., N_n\} \) and \( N_T \) contains at least one CIR-node. Table 1 summarizes the rules that define legal and illegal reorderings.

<table>
<thead>
<tr>
<th>Type of optimization</th>
<th>Type of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra-region reordering</td>
<td>JLS</td>
</tr>
<tr>
<td>inter-region reordering</td>
<td>JMM</td>
</tr>
</tbody>
</table>

Table 1. Reordering rules for intra- and inter-region reordering optimizations. The rules are defined in the Java Language Specification (JLS) [7], the Java Memory Model (JMM) [16], and in the subsequent definition of Concurrency Invariance (CI).

Concurrency invariant intra- and inter-thread actions can be moved into concurrent regions. Concurrency-invariance is defined as follows: An IR-node \( N \) that is defined in a home region \( C_H \) and is concurrency invariant to a set of target regions \( \{C_{T1}, ..., C_{Tn}\} \) is defined as: \( N \xrightarrow{C_{inv}} \{C_{T1}, ..., C_{Tn}\} \). If \( N \) is concurrency invariant to \( C_T \), \( N \) can either be moved into \( C_T \) by reordering \( N \) with cStart (if \( N \) is defined before (in program order) \( C_T \)), or by reordering \( N \) with cEnd (if \( N \) is defined after (in program order) \( C_T \)). An IR-node \( N \) is concurrency invariant if and only if:

- \( N \) does not change the control flow;
- moving \( N \) from \( C_H \) to \( C_T \) preserves the intra-thread semantics of \( C_H \) and \( C_T \);
- moving \( N \) from \( C_H \) to \( C_T \) preserves the inter-thread semantics of \( C_H \) and \( C_T \).
4.2.1 Control flow

Instructions that change the control flow are not concurrency invariant, since the reordering potentially changes the condition under which a piece of code can be executed in parallel. Consider the example shown in Figure 7, which depicts the effects of reordering the cStart CIR-node with an if IR-node. Figure 7(a) shows a sequential region (SR) that conditionally executes a concurrent region (CR). I.e., the then part contains a concurrent region and the else part contains sequential regions. Figure 7(b) shows the result of moving the if-node into the concurrent region. In Figure 7(b) the child thread executes the then part but potentially also the else part of the if-statement. Such a behavior is, in general, illegal since the else part must be executed by the same thread as the SR. For example, the else part can write to a global variable that is read after the if statement by the thread that executes the SR. However, if the else part is executed by a separate child thread, the code transformation shown in Figure 7(b) introduces a data race that is not present in the original version of the program.

![Effects of reordering branch instructions.](image)

Figure 7. Effects of reordering branch instructions.

4.2.2 Preserving intra-thread semantics

Intra-thread actions are actions that are not visible to other threads. For example, adding two local variables is an intra-thread action, since local variables are stored on the stack and the stack is private to each thread. As a result, changes to local variables that are performed in a concurrent region are not visible outside the concurrent region. The runtime system initializes live [18] local variables that are used in a concurrent region to the values of the corresponding variables in the parent thread.

Concurrent invariant intra-thread actions Figure 8 illustrates two sequential regions, one concurrent region, as well as potentially concurrency invariant reorderings (RO1, RO2, and RO3) and reorderings that are always illegal (RO4 and RO5). The conditions that specify if an intra-thread action is concurrency invariant are defined in Table 2. RO4 and RO5 are, in general, illegal for intra-thread actions (see Section 8 in [16]). The bidirectional arrows for RO4 and RO5 indicate that reorderings are prohibited in either direction.

![Reordering constraints for standard compiler optimizations in CIR.](image)

Figure 8. Reordering constraints for standard compiler optimizations in CIR.

Assume that I1 defines a local variable l1 and I2 defines a local variable l2. Furthermore, assume that I1 and I2 are intra-thread actions that do not change the control flow and cannot throw an exception.

RO1: I1 \xrightarrow{\text{inv}} \{C\} if \(\text{def}(l_1)\) is not used in SR2. Note that changes to local variables that are performed by the child thread are not visible to the parent thread.

RO2: I1 \xrightarrow{\text{inv}} \{C\} if \(\text{def}(l_1)\) is not used in C. Killing a live \(\text{def}(l_1)\) changes the intra-thread semantics of the child thread.

RO3: I4 \xrightarrow{\text{inv}} \{C\} if \(\text{def}(l_2)\) does not kill another \(\text{def}(l_2)\) in C. Killing a live \(\text{def}(l_2)\) changes the intra-thread semantics of the child thread.

Table 2. Concurrency invariant operations: local variables

The JIT compiler uses standard liveness analysis [18] to determine the conditions for RO1, RO2, and RO3.

4.2.3 Preserving inter-thread semantics

The section discusses concurrency invariant inter-thread actions. Inter-thread actions are normal loads and stores, volatile loads and stores, monitorenter and monitorexit bytecodes, wait() and notify(), as well as starting and joining threads.

Normal loads and stores Reordering RO1, RO2, and RO3 from Figure 8 are concurrency invariant to C if the conditions listed in Table 3 hold. Assume that I1 and I4 do not change the control flow and that a load instruction (ld) loads the value from a location on the heap (o.f) into a local variable l and that a store instruction (st) stores the value of v to the field o.f.
4.3 Concurrent region merging

This section discusses the conditions that enable the JIT compiler to merge concurrent regions. Merged concurrent regions have a coarser PTG compared to unmerged concurrent regions.

4.3.1 Merging non-merged concurrent regions

Assume there are two concurrent regions \( C_1 \) and \( C_2 \) and that \( C_1 \) as well as \( C_2 \) correspond to a single concurrent method in the source code. I.e., \( C_1 \) and \( C_2 \) are not merged with other concurrent regions. Furthermore, assume that \( C_1 \) is defined in basic block \( BB_1 \) and \( C_2 \) is defined in \( BB_2 \). \( C_1 \) and \( C_2 \) can be merged if and only if:

1. \( C_1 \) and \( C_2 \) are in the same basic block and there is no instruction between \( C_1 \) and \( C_2 \)
2. all instructions between \( C_1 \) and \( C_2 \) are concurrency invariant to either \( C_1 \) and/or \( C_2 \)
3. \( BB_1 \) dominates \( BB_2 \) and
   (a) there is no \( \text{CIR-node} \) annotated method on any path from \( C_1 \) to \( C_2 \) and
   (b) there is no synchronization action in a sequential region (or concurrent region that is compiled to sequential code) that is on a path from \( C_1 \) to \( C_2 \).

\textbf{Condition 1:} If there is no instruction between \( C_1 \) and \( C_2 \), \( C_1 \) and \( C_2 \) are contiguous concurrent regions. Merging two contiguous concurrent regions is trivially possible. Contiguous concurrent regions must be contained in the same basic block. Otherwise, there would be an instruction (e.g., a \texttt{goto}) between \( C_1 \) and \( C_2 \).

\textbf{Condition 2:} If all instructions between \( C_1 \) and \( C_2 \) are concurrency invariant to either \( C_1 \) and/or \( C_2 \), the instructions can be moved into \( C_1 \) or \( C_2 \). As a result, the concurrent regions are contiguous and therefore mergeable.

\textbf{Condition 3:} \( BB_1 \) must dominate \( BB_2 \) so that the merged concurrent region can safely be issued at \( BB_2 \). If \( BB_1 \) does not dominate \( BB_2 \), the concurrent region that is defined in \( BB_2 \) is eventually never executed. Consider the examples as shown in Figure 9. Figure 9(a) illustrates the original program. The arrows in Figure 9 indicate transitions between basic blocks that are triggered in a sequential region. I.e., the transition from \( BB_2 \) to \( BB_3 \) and \( BB_2 \) to \( BB_4 \) is not determined in \( C_2 \).

Figure 9(b) shows a variant of the original program in which the concurrent regions \( C_1 \) and \( C_2 \) are merged. \( BB_1 \) and \( BB_2 \) can be combined into a single basic block. As a result, \( C_1 \) and \( C_2 \) are contiguous concurrent regions and can therefore be legally merged. Figure 9(c) shows an example in which \( C_2 \) is merged with \( C_3 \) and \( C_2 \) is merged with \( C_4 \). Note that either merging \( C_2 \) with \( C_3 \) or \( C_2 \) with \( C_4 \) results in an illegal program, since \( C_2 \) is then not guaranteed to be executed in any path from \( BB_1 \) to \( BB_3 \). Finally, Figure 9(d) shows a variant of the original program in which \( C_2 \) is merged with \( C_5 \). This code transformation is legal if \( C_3 \) and \( C_4 \) are compiled to parallel code. Since \( BB_2 \) dominates \( BB_5 \), both concurrent regions are guaranteed to be executed. Merging \( C_2 \) with \( C_5 \) determines a particular schedule of execution.

\textbf{Condition 3.a:} Two concurrent regions \( C_1 \) and \( C_2 \) cannot be merged if there is a synchronization method on any path in the control flow graph (CFG) from \( C_1 \) to \( C_2 \).
shows the effects of such a merging. Figure 10(a) shows the original program. Figure 10(b) shows a variant of the original program in which $C_1$ and $C_2$ are merged ($C_{12}$) and the execution of $C_{12}$ is issued after the synchronization method. Such a code transformation introduces a new behavior, since there is no guarantee that the effects of $C_1$ are visible to other threads before $C_2$ starts executing. Similarly, the code transformation as shown in Figure 10(c) potentially introduces a new behavior, since there is no guarantee that the effects of $C_1$ are visible to other threads before $C_2$ starts executing. Recall that these visibility guarantees are provided by the cSync CIR-node.

**Condition 3.b:** If there is a synchronization action (e.g., a volatile memory operation) in a sequential region on any path in the CFG from $C_1$ to $C_2$, merging $C_1$ with $C_2$ potentially introduces new behavior, since the ordering guarantees of the original program are changed. For example, consider the program as given in Figure 10. Assume that $C_1$ contains a volatile store to a field ($x = 1$) and $C_2$ contains the following volatile store ($x = 0$). Furthermore, assume that the cSync CIR-node is replaced by the following while statement: while ($x == 0$). Initially, $x = 0$. The original program, as illustrated in Figure 10(a) guarantees that both concurrent regions as well as any region that follows the volatile store to field $x$ is moved after the volatile read in the while loop. If $C_1$ and $C_2$ are merged as illustrated in Figure 10(b) the merged concurrent region will never be executed, since the volatile store to field $x$ is moved before the volatile read in the while loop. If both behaviors are not possible in the original program, such a code transformation is illegal.

### 4.3.2 Merging merged concurrent regions

Assume there are two merged concurrent regions $C_x$ and $C_y$. $C_x$ and $C_y$ can be merged if and only if every concurrent region that is merged into $C_x$ is mergable with every concurrent region that is merged into $C_y$.

### 4.4 Merging example

Figure 11 shows concurrent region merging in CIR. Figure 11(a) shows the original program and Figure 11(b) depicts the CIR of the merged version. In particular, function calls B() and C(), which are contained in a separate concurrent region in Figure 11(a) are merged into one concurrent region in Figure 11(b). Note that the cBarrier CIR-node is required to guarantee sequential consistency for data race-free programs.

```
1 call A
2 call B
3 call C
4 call B
5 call C
6 call C
7 call C
8 call C
```

![Figure 10. Merging of concurrent regions with cSync](image)

**Figure 11.** Merging of concurrent regions.

### 5. Profiling system extension

The existing profiling infrastructure must be extended to obtain two necessary performance characteristics of concurrent regions that enable effective merging of concurrent regions: the execution time ($T_{exec}$) and the number of invocations ($I$) of a concurrent region. $T_{exec}$ is important, because the overhead from using a separate thread to execute the concurrent region must be smaller than the performance gain due to a parallel execution to speedup the application. $I$ is important since it represents the number of tasks that are generated at runtime. A large number of tasks provides good load balance but introduces a running-time overhead.

The Jikes Research Virtual Machine (Jikes RVM) sampling profiler can only provide an estimate of how much time the application spent in a particular method ($T_{total}$). $T_{total}$ does not enable the AOS [3] to draw inferences about $T_{exec}$ and $I$. A large $T_{total}$ can have two reasons: First, $T_{exec}$ is indeed long (and executed infrequently). However, a large $T_{total}$ can also result from a frequently invoked method with a short $T_{exec}$. The sampling technique of the Jikes RVM is also unable to count the number of invocation $I$ of a concurrent region. To precisely determine $T_{exec}$ as well as the $I$, we add an instrumenting profiler to the Jikes RVM.

#### 5.1 Concurrent region instrumentation

The JIT compiler instruments a concurrent region as illustrated in Figure 12. In particular, the JIT compiler uses the rdtsc instruction to measure the execution time. The execution time is only recorded at certain intervals. The if statement in line 2 checks if the time since the last record (thread.rec) is larger than THRESHOLD. To perform this check, each thread has a field, rec, that stores the time stamp counter when the last sample was recorded. The rec field is initialized to 0 when a thread is created. If the if statement in line 2 evaluates to true, the execution time of the concurrent region is recorded and the rec field is updated. Otherwise, the sample is not recorded.
5.2 Maintaining performance samples

Each thread stores the gathered samples to a data structure that is private to the thread. The bytecode index of the concurrent region and the signature of caller of the concurrent region form the key for a hash map that holds the reference to a circular buffer that stores the performance samples. If the circular buffer collected a certain amount of new samples (report threshold) the arithmetic average of the samples is reported to the global database, which collects the samples of all threads. The report threshold starts at 1 and increases up to 100 in strides of 10. Such a reporting strategy ensures that samples of newly discovered concurrent regions are reported immediately and frequently to the global database. The samples of concurrent regions that have been profiled for a longer time are reported less frequently. Each thread maintains the number of concurrent region invocations in a similar manner.

There is a second condition that must be fulfilled so that a thread reports a sample to the global database. The arithmetic average over the collected samples must differ by at least by 30% compared to the last reported average. Reporting similar samples provides no new information. We choose 30% based on empirical evaluation.

If a thread commits a new sample to the global database, the committing thread wakes up another thread (which belongs to the AOS) that evaluates the new data asynchronously. The next section describes the evaluation process of the samples in detail.

6. Adaptive optimization system extension

This section describes the parallel code execution optimization (PCEO) algorithm. The PCEO algorithm aims at finding a setting to execute concurrent regions efficiently. The PCEO algorithm determines if a concurrent region is (i) executed sequentially, (ii) executed in parallel, and (iii) is merged with other concurrent regions. The PCEO algorithm uses profile information collected at run-time to determine the setting of the concurrent regions at method-level granularity. Figure 13 shows the PCEO algorithm in detail.

The PCEO algorithm takes the CIR, the new profile information, and the merging function (see later this section) as an input. The PCEO algorithm uses two thresholds: the serialization threshold (T_SER) and the merging function. T_SER represents the overhead that is associated with a parallel execution. I.e., T_SER corresponds to the execution time of a parallel task at which a parallel execution of the task is equally fast as the sequential execution. Consequently, T_SER can be used to determine if a concurrent region is executed sequentially or in parallel. Our experimental evaluation yields a value of 10,000 cycles for T_SER.

The PCEO algorithm first computes the oversubscription factor (OS), (line 4), which describes the relation between the number of parallel concurrent region invocations and the number of available cores. An oversubscription factor larger than 1 means that there are more parallel concurrent region invocations than available cores. The algorithm completes if it finds a new merging configuration (line 40) or if the work list is empty. From line 9 to line 20, the algorithm handles concurrent regions that cannot be merged with other concurrent regions. Line 10 to line 15 check if the execution time of the concurrent region is smaller than T_SER. Furthermore, the algorithm checks if the concurrent region is currently compiled to parallel code. If so, the PCEO algorithm issues a serializing recompilation (line 12 and line 13).
Input: CIR, inf(profiles), mf(merge function)  
Output: CIR

int OS = inf.cRegionInvocations / numProcessors;
Worklist workList = inf.getCRegionProfiles();
inf.recompile = false;
while (true) {
    CRegion r = workList.get();
    if (!r.hasMergeCandidates()) {
        if (r.execTime < T_SER) {
            r.isParallel = false;
inf.recompile = true;
        } else if (r.execTime > T_SER) {
            if (!r.isParallel) {
                r.isParallel = true;
inf.recompile = true;
            }
        } else {
            boolean merge = shouldMerge(r, OS);
            boolean demerge = shouldDemerge(r, OS);
            if (merge) {
                r.mergeCandidates.removeAll(r.mergedWith);
                r.isParallel = false;
            } else if (demerge) {
                r.mergedWith.clear();
inf.recompile = true;
            } else {
                workList.removeAll(r.mergedWith);
            }
        }
    } else {
        handleRemainingRegions();
    }
}

Figure 13. Parallel code execution optimization algorithm.

From line 15 to line 19 the algorithm checks if the execution time of the concurrent region is larger than T_SER. If the concurrent region is compiled to sequential code, the PCEO algorithm issues a parallelizing recompilation.

Line 21 to line 48 handle concurrent region merging/demerging. shouldMerge() and shouldDemerge() return a boolean value that indicates if a concurrent region is merged or demerged. Both functions are shown in Figure 14 and Figure 15, respectively. If the algorithm decides to merge concurrent regions, the algorithm first removes all concurrent regions from the list of potential merging candidates that are already merged with the concurrent region (line 25). In the next step, the algorithm sets the isParallel flag of the current concurrent region to false. The reason is that the execution time of the (merged) concurrent region can be lower than T_SER. From line 27 to line 41, the algorithm iterates over the merging candidates. For each merging candidate, the PCEO algorithm (i) removes the merging candidate from the work list, (ii) adds the candidate to the existing set of merged regions, (iii) clears the existing merging configuration of the merging candidate, and (iv) sets the isParallel flag to false. Line 34 to line 35 compute the execution time of the merged concurrent region. If the merged execution time is larger than T_SER, the merged concurrent region is compiled to parallel code (line 38 and line 39).

Line 43 to line 46 handle the demerging. Demerging is currently done by recompiling all merged regions to either sequential or parallel code. Line 45 sets the recompile flag to true. As a result, all concurrent regions, including the concurrent regions that were originally merged with r, are handled by handleRemainingRegions() in line 50. This function simply checks for each non-merged concurrent region if the execution time is above or below T_SER and sets the isParallel flag accordingly.

Figure 14 shows the shouldMerge() function, which returns a boolean value that indicates whether the concurrent region should be merged with other concurrent regions. Line 2 examines the execution time of the concurrent region. If the execution time is below T_SER, shouldMerge() returns true. Otherwise, line 4 computes the merge function. The returned value b is passed to the withinTolerance() function, which checks if the current oversubscription factor (OS) is within +/- 10% of the value of the merge function. This check avoids frequent recompilations that have only minor effects. Furthermore, if the current oversubscription factor is larger than the computed merge function value, shouldMerge() returns true in line 6.

Figure 15. Algorithm that calculates demerging condition.
Figure 15 shows the `shouldDemerge()` function. Line 2 computes the merge function. Similar to the `shouldMerge()` function, line 3 checks if the current value of the OS is +/- 10% the value of the computed merge function value (b). If the current oversubscription factor is smaller than b, `shouldDemerge()` returns true in line 4.

**Merge function** The merge function defines if concurrent region merging or demerging is applied. The merge function can be supplied at JVM startup. If no merge function is provided, the JVM uses the default merge function that is illustrated in Figure 16.

![Figure 16. Merge function example. The merge function divides the merge-region from the demerge-region.](image)

The x-axis in Figure 16 shows the *overhead factor* (OH), which is defined as follows:

\[
OH = \frac{T_{SER}}{T_{exec}}
\]

The overhead factor is a metric for the impact of the parallel overhead on the execution time of a concurrent region. i.e., a low overhead factor indicates that the concurrent region incurs a low parallel overhead compared to its execution time. An overhead factor larger than 1 implies that execution time of the concurrent region is shorter than the parallel overhead. As a result, such a concurrent region is always merged with another concurrent region, or the concurrent region is serialized. The y-axis illustrates the oversubscription factor (OS). Recall that an oversubscription factor of 1 means that the JVM generates 1 task for each available core. The JVM computes the overhead factor for every concurrent region and the oversubscription factor for every method that contains at least one concurrent region.

The main idea behind the merge function is to provide the user of the JVM with the opportunity to specify the efficiency at which parallel code is executed *without* having to adapt the source code. We define the efficiency of a parallel execution as the number of generated tasks in relation to the potential performance gain.

**Example** Assume concurrent region \( C \) has an \( OH = 0.05 \) and that the \( OS \) of the method in which \( C \) is defined has \( OS = 0.1 \) (dot in Figure 16). Furthermore, assume that \( C \) is merged with other concurrent regions. The merge function as depicted in Figure 16 triggers a demerging recompilation, since \( C \) yields a spot in the demerge region.

### 7. Evaluation

The section presents the experimental evaluation of the overhead that comes from concurrent region profiling as well as the performance evaluation using the Java Grande Benchmark Suite [22]. The performance evaluation uses our prototype implementation of the extensions presented in Section 4, Section 5, and Section 6 in the Jikes RVM [2].

#### 7.1 Experimental setup

All experimental results are obtained from a system that is equipped with 32 Intel Xeon E7-4830 cores @2.13 GHz and 64 GB of main memory. The system runs Ubuntu 11.10 (64-bit) with kernel 3.0.0. We use Java version 1.6.0_24, a minimum (and maximum) heap size of 2 GB, and four parallel garbage collector threads. The Jikes RVM is compiled in the FullAdaptiveCopyMS configuration. We use this setup since other setups (e.g., the production configuration) result in an unstable behavior of the Jikes RVM.

#### 7.2 Profiling overhead

The running time overhead of execution time profiling is determined by using a synthetic benchmark. The synthetic benchmark consists of a loop that contains one concurrent region that in turn contains one method call. The loop body is executed \( 10^7 \) times to get stable execution times. The baseline for the profiling overhead evaluation is the execution time of an unprofiled version of the synthetic benchmark that is compiled to sequential code. Sequential code is better suited to measure the profiling overhead, since sequential code does not incur parallel overheads. The results are therefore more stable. To measure the profiling overhead of parallel threads *without* including the parallel overhead, the parallel version of the synthetic benchmark starts Java threads that execute the same concurrent region that is compiled to sequential code. To ensure that all threads execute the main benchmark loop at the same time the synthetic benchmark has a barrier before the main benchmark loop.

The performance results presented in this section are gathered by executing the benchmarks 20 times in the same JVM instance. We use 10 warmup runs and 10 profile runs. The warmup runs are not included in the results to measure steady-state performance. The presented results are the arithmetic average over the 10 profile runs. The error bars indicate the standard deviation.

The baseline is compared against an instrumented (profiled) version of the synthetic benchmark that is also compiled to sequential code. To measure the impact of the *record*
interval on the profiling overhead, the synthetic benchmark is executed using different values for the record interval (0 cycles to $10^6$ cycles). The reported profiling overhead is the sum of the overhead from storing a sample into the thread-local database and the overhead of committing a thread-local sample to the global database.

The main source of parallelism in the Java Grande benchmarks are parallel loops. The Java versions implement parallel loops by assigning each thread an equally-sized chunk of the iteration space of the loop. The versions using concurrent methods are parallelized by outlining the body of each parallel loop to a separate method that is annotated with `@Concurrent`. To provide the opportunity to merge concurrent regions, we manually unroll parallel loops. We must unroll parallel loops manually, since the loop unrolling pass in the JIT compiler of the Jikes RVM produces unstable code. Note that the manual unrolling of parallel loops is only necessary due to this missing standard optimization in the Jikes RVM. Production JVMs like Hotspot perform loop unrolling and it is easy for the JIT compiler to identify concurrent regions that are in a loop. The JIT compiler can force loop unrolling (and therefore generate mergable concurrent regions) if the overhead factor of the concurrent region suggests to merge the concurrent region. We unroll parallel loops of all benchmarks by a factor of 8. As a result, the PCEO algorithm can coarsen the given parallel code granularity by a factor of 8.

The performance results presented in this section are gathered by executing the benchmarks 50 times in the same JVM instance. We use 10 warmup runs and 40 profile runs. The warmup runs are not included in the results to measure steady-state performance. The presented results are the arithmetic average over the 40 profile runs. The error bars indicate the 95% confidence interval.

Each benchmark is executed with the same Jikes RVM configuration. In particular, we use a profiling interval (THRESHOLD) of 1 million cycles, a serialization threshold of 10,000 cycles, and the following function, which describes the merging and demerging threshold.

$$M = OH^{-\frac{1}{X}} - Y$$

We choose 0.99 for $X$ and for $Y$. Figure 16 illustrates the merging function. We use this merging function for the following reasons: First, the merging function provides a high oversubscription factor (number of tasks per core) for a low overhead factor. As a result, the system is well balanced if the parallel overhead does not dominate the execution time of the application. Recall that an overhead factor of 0 means that there is no parallel overhead. An overhead factor of 1 means that the parallel overhead is equal to the execution time of the task. Second, the oversubscription factor decreases exponentially with an increasing overhead factor. As a result, the efficiency of the system is well maintained. The JVM is equipped with this merge function by default.

Figure 18 shows the performance results obtained by our automated approach to determine the PTG compared to three different Java versions. The Java versions differ regarding the PTG: The PTG in Figure 18(a) is chosen such that the benchmark yields best performance. To find the best-performing PTG, we manually try different PTGs. I.e., for each parallel loop, we divide the iteration space of the loop into 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 equally-sized parallel tasks that are executed in parallel by 32 threads. Figure 20 in Appendix B illustrates the performance results for the manual PTG performance runs. The sequential execution times of the evaluated benchmarks that are the baseline (a value of 1 on the y-axis) for Figure 20 are as follows: `crypt`: 4.9s, `sor`: 24.2s, `lu`: 1.4s, `matmult`: 1.6s, and `montecarlo`: 24.2s. The PTG that yields the best performance for a benchmark is the baseline in Figure 18(a) (512 tasks for `crypt`, 1024 tasks for `sor`, 64 tasks for `lu`, 64 tasks for `matmult`, and 512 tasks for `montecarlo` – in this order). It is easy to see that finding a good PTG is difficult.

The y-axes in Figure 18 compare the performance of the Java versions against our approach. A value of 1 on the y-axis means that our approach is equally fast compared...
to a Java version. A value smaller than 1 means that our approach is slower than standard Java; a value larger than 1 means that our approach is faster. The error bars indicate the 95% confidence interval over the 40 performance runs. To enable a fair comparison between our approach and the Java versions, our approach and the all Java versions use the same thread pool implementation. The performance results of our approach include the overhead from dynamic profiling and dynamic recompletions that adapt the PTG. Table 4 in Appendix A shows the number of generated parallel tasks for each benchmark.

Figure 18(a) shows that our approach performs slightly slower than the manually optimized Java version. This is not surprising, since the manually optimized version uses offline profiling to determine the PTG. The performance penalty due to dynamic recompletion is insignificant, since the number of recompletions imposed by adapting the PTG is small. On average less than 30 recompletions are needed to determine the final PTG.

Figure 18(b) shows the performance of our approach compared to the Java version that assigns the iterations of parallel loops to 8 parallel tasks. Our approach outperforms Java for crypt, series, matmult, and montecarlo, since parallelism is underspecified for a 32-core system. Our approach performs slower for the sorr and the lu benchmark, since our approach cannot effectively remove all overspecified parallelism. The reason is that parallel loops are manually unrolled by a factor of 8. If merging all 8 concurrent regions results in a PTG where parallelism is overspecified, our approach cannot remove more parallelism. This is, however, an implementation feature of the Jikes RVM. If the Jikes RVM can determine the unroll factor, e.g., an unroll factor larger than 8, more parallelism can be removed.

Figure 18(c) illustrates the performance of our approach compared to the Java version which splits the iteration space of parallel loops into 512 parallel tasks. Our approach performs faster for the sorr, lu, and the matmul benchmark, since parallelism is overspecified in for these benchmarks.

8. Related work

There exists a large body of related work in the area of parallelizing compilers and compiler optimizations for parallel programs. For example, Lee et al. [14] present compiler algorithms for explicit parallel programs. The authors introduce a concurrent static single assignment form (CSSA) for programs containing cobegin/coend, and post/wait statements. In another paper, Lee et al. show how to apply constant propagation on CSSA [13]. Sarinivasan et al. present an SSA form for explicit parallel programs [23].

The main difference of [13, 14, 23] to CIR is that CIR can be easily integrated into existing JIT compiler infrastructures. For example, CIR can be transformed to SSA without having to change SSA construction. If code reorderings are kept local to a concurrent region (which can be done by giving the CIR-nodes a property that disallows all reorderings), no modifications to existing optimization passes are needed. Furthermore, the compiler transformations in [14, 23] generate sequentially consistent (SC) code [12]. SC is more restrictive than the JMM [16] and therefore forbids many effective optimizations. SC Java code is 10% - 26% slower than Java code compiled according to the JMM [24].

Other examples of parallel program analysis include an optimal bit vector analysis [11] for explicit parallel programs that allows to compute reaching definition, definition-use chains, or analysis for code motion. Grunwald et al. present data flow equations for explicitly parallel programs [8]. Finally, Sarkar presents analysis techniques that are based on a parallel program graph [20]. These efforts all perform program analysis in a static compiler. A JIT compiler must generate high quality code in a short time. To the best of our knowledge, we present the first JIT compiler that operates on an explicit parallel IR that can produce high quality code. Pramod et al. present a technique that allows to reuse compiler optimizations that are designed for sequential code unmodified to parallel code [9]. This work is similar to our work, since existing optimizations can be applied to CIR without any modifications. However, the algorithms presented in [9] guarantee sequential consistency, which is a more restrictive memory model that the JMM.

There are several OpenMP compilers that optimize in and around parallel regions. E.g., Satoh et al. present a reaching definition analysis, memory synchronization analysis and loop data dependence analysis for parallel loops [21]. Tian et al. present multi-entry threading, a compilation technique that allows to perform optimizations across thread boundaries [25], Zhang et al. exploit meta information provided by OpenMP directives to analyze barrier synchronization [27].

Cilk [6] extends the C/C++ programming language with parallel calls and a construct to synchronize parallel calls. The semantics of concurrent statements correspond to the semantics as proposed in Cilk. The main difference to Cilk is that our approach integrates concurrent regions into a runtime system that allows dynamic optimizations such as profile-based recompletion [26]. The current Cilk runtime cannot perform such optimizations. Rugina et al. [19] present a flow-sensitive, context-sensitive, inter-procedural pointer analysis algorithm for Cilk programs.

X10 [5] provides language-based explicit parallelism. Zhao et al. [28] present a compiler framework that aims at reducing the task-creation and termination overhead. The optimizations are similar to concurrent region merging. However, the authors use a static compilation approach, which makes the underlying runtime oblivious of the parallel constructs. The authors in [4] present an load elimination algorithm for X10 programs. Load elimination is a reordering optimization and the algorithm presented in [4] supports the elimination of load across async statements.
Most approaches to describe parallelism in object-oriented languages are library-based. For example, Intervals [17] provide an abstraction to explicitly specify the execution order of tasks. Parallel Java [10] provides OpenMP and MPI constructs for Java. Leiden et al. present a task parallel library for .NET[15]. Library-based approaches for dynamically compiled languages have the drawback that the JIT compiler is unaware of the parallel constructs. As a result, the JIT compiler has less information to optimize parallel code.

9. Conclusion

This paper presents a novel approach for an automatic, dynamic, and feedback-directed determination of the parallel task granularity. Our approach is based on two principles that are not present in the current Java Runtime Environment. The first principle is the separation of the declaration of parallelism from its execution. This principle is provided by concurrent calls and allows the system to shift the responsibility to determine how parallel code is executed from the application/library level to the JVM level. The second principle is the concurrency-awareness of the JVM, which enables the JVM to gather profile information that is relevant to optimize the performance of parallel code.

Manually tuning the parallel task granularity is tedious, time-consuming, and often impossible in practice. Our approach represents an important step towards automating this important task.

References


A. Concurrency invariance of inter-thread actions

Volatile load/store operations Volatile load operations are never concurrency invariant, since making volatile loads concurrency invariant breaks the ordering guarantees provided by volatile semantics. Consider the example in Figure 19. Figure 19(a) shows the original program and Figure 19(b) shows a version of Figure 19(a) in which the JIT compiler moves the volatile load of \(y\) to Thread 2 into the concurrent region. The behavior \(r1 == 3\) and \(r3 == 0\) is illegal, since volatile load/store operations cannot be reordered. Consequently, if \(r1 == 3\) Thread 1 must have executed line 2 and as a result also line 1. However, Figure 19(b) allows \(r1 == 3\) and \(r2 == 0\), since the code in the concurrent region is executed asynchronously with the code that follows the concurrent region. In particular, line 9 in Figure 19(b) can be executed prior to line 5. Such a behavior is illegal in the original code. Reordering a volatile load with the cEnd CIR-node (e.g., line 9 in Figure 19(a)) causes an equivalent behavior for a concurrent region that follows line 9.

Moving volatile stores into a concurrent region introduces, in general, a program behavior that is not possible in the original program and is therefore illegal. Similar to the example shown in Figure 19, moving a volatile store into a concurrent region breaks the happens-before relationship between the volatile store and the preceding (in program order) store instructions. The reason is the asynchronous execution of a concurrent region.

monitorenter and monitorexit The bytecodes monitorenter and monitorexit implement the synchronized statement. As specified in the JLS, monitorenter acquires the intrinsic lock of an object and monitorexit releases the intrinsic lock. Moving either of both bytecodes into a concurrent region changes the inter-thread semantics of the child thread and the parent thread, since acquiring/releasing the lock is then performed by a different thread. As a result, both bytecodes are not concurrency invariant.

wait(), notify(), and notifyAll() The functions wait(), notify(), and notifyAll() are low-level communication primitives. Since wait() requires that the calling thread owns the monitor of the object on which the thread calls wait(), wait() is not concurrency invariant. If the JIT compiler moves the wait() call into a concurrent region and the concurrent region is executed by the child thread, the child thread does not own the monitor. Such a code transformation raises an exception that does not occur in the original program. notify() and notifyAll() are not concurrency invariant for similar reasons as wait(). Both calls require that the thread which performs the call owns the corresponding monitor.

Starting and joining threads Starting and joining threads is not concurrency invariant, since both operations are implemented as native function calls that cannot be inlined.

B. Additional performance results
Figure 20. Manual tuning of parallel task granularity of the Java Grande benchmark Suite on the 32-core system. The x-axes show the number of parallel tasks used to process a parallel loop. The y-axes present the speedup over a sequential execution. The error bars indicate the 95% confidence interval over the 40 performance runs.

Table 4. Number of thousand parallel tasks for our approach and the standard Java versions.