Online Feedback-Directed Optimizations for Parallel Java Code

Doctoral Thesis

Author(s):
Noll, Albert

Publication date:
2013

Permanent link:
https://doi.org/10.3929/ethz-a-009997426

Rights / license:
In Copyright - Non-Commercial Use Permitted

Diss. ETH No. 21080

Online Feedback-Directed Optimizations for Parallel Java Code

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Albert Noll
Dipl.-Ing., TU Graz
born April 20, 1981
citizen of Austria

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Samuel P. Midkiff , co-examiner
Prof. Dr. Walter Binder, co-examiner

2013

Abstract

Multi-core processors are nowadays standard in servers, desktop computers, and mobile devices. To make applications exploit the new hardware resources that are provided by the large number of cores, programmers must write parallel code that runs concurrently on the available cores. State-of-the-art parallel programming techniques require the programmer to define concurrent code regions and the conditions (concurrency condition) under which concurrent code regions are executed in parallel. If the concurrency conditions are set unlucky, the performance of parallel code can drop way below its potential. Moreover, it is even possible that parallel code performs slower than a corresponding sequential version, if the overhead that is associated with every parallel execution exceeds the performance gain of a parallel execution.

This thesis examines an automated, online, and feedback-directed approach to determine the concurrency conditions under which concurrent code regions are effectively executed in parallel. The approach is automated, since the runtime system determines the concurrency conditions based on the knowledge which code regions can be executed concurrently and requires no additional input. The approach is online, since the concurrency conditions are determined at run-time. Traditional parallel programming models require a static specification of the concurrency conditions. Finally, the approach is feedback-directed, since the concurrency conditions are constantly re-evaluated at run-time.

The thesis explores the determination of the concurrency conditions in the context of the Java platform. The extension of the concurrency interface between the Java source language and the Java Virtual Machine (Java VM) enables the Java VM to determine the concurrency conditions automatically at run-time. The concurrency interface defines the available knowledge of the concurrency structure of an application in the Java VM and the just-in-time compiler (JIT compiler). The standard Java concurrency interface is extended by two constructs: concurrent statements and synchronization statements. A concurrent statement *may* be executed in parallel with the subsequent statement. A synchronization statement enables the synchronization of concurrent statements. Concurrent statements and synchronization statements are represented explicitly in the intermediate representation of the JIT compiler. Such an explicit representation enables the JIT compiler to instrument a concurrent statement so that the profiling system can precisely determine the behavior of the concurrent statement. Based on the measured behavior, the JIT compiler compiles a concurrent statement to (i) sequential code, (ii) parallel code, or (iii) merges a concurrent statement with another concurrent statement and thereby changes the parallel code granularity of the application.

This thesis presents the design, implementation, and evaluation of the extended concurrency interface for the Java platform. More precisely, the implications of such a system for the Java language, the Java VM, the JIT compilers, and the Java Memory Model (JMM) are discussed in detail. We obtain experimental results from a prototype implementation and show that our automated determination of the concurrency conditions results in competitive performance compared to hand-optimized code. More specifically, if the hand-optimized version finds good concurrency conditions, our approach performs equally fast or slightly slower (15%). If, however, the concurrency conditions of the hand-optimized code are set unlucky, our approach is up to 4X faster.

Zusammenfassung

Mehrkernprozessoren findet man in Servern, Desktop-Computern und mobilen Geräten. Die zusätzlichen Prozessorkerne stellen neue Hardware Ressourcen zur Verfügung. Einzelne Applikationen können diese Ressourcen nur dann effizient nutzen, wenn paralleler Code nebenläufig auf den Mehrkernprozessoren ausgeführt wird. Auch bei der Anwendung modernster Programmiertechniken müssen die Kriterien zur parallelen Ausführung des Programms von Hand definiert werden. Diese Kriterien bezeichnen wir als Parallelisierungskriterien. Werden die Parallelisierungskriterien falsch spezifiziert, kann die Ausführungszeit eines parallelen Programms langsamer sein als eine sequentielle Ausführung. Dies kann auftreten, wenn der Mehraufwand, der mit jeder parallelen Ausführung verbunden ist größer ist als der Leistunggewinn, der von einer parallelen Ausführung herrührt.

Das Ziel dieser Dissertation ist eine Methode zu entwickeln, welche die Parallelisierungskriterien automatisch bestimmt, so dass paralleler Code effizient ausgeführt wird. Die Parallelisierungskriterien werden zur Laufzeit bestimmt und passen sich dem Verhalten des Programms an. Diese Dissertation beschreibt eine solche Methode für die Java Umgebung. Dazu wird die Schnittstelle zwischen dem Java Quellcode und der Java virtuellen Maschine (Java VM) um zwei Anweisungen erweitert: potentiell parallele Anweisungen und blockierende Anweisungen. Potentiell parallele Anweisung können, müssen aber nicht parallel ausgeführt werden. Blockierende Anweisungen halten die Programmausführung an bis alle potentiell parallelen Anweisungen ausgeführt sind. Potentiell parallele Anweisungen sowie blockierende Anweisungen werden explizit im Just-In-Time (JIT) Compiler dargestellt. Diese explizite Darstellung ermöglicht das Laufzeitverhalten von potentiell parallelen Anweisungen genau zu analysieren und die Parallelisierungskriterien entsprechend zu optimieren. Der JIT Compiler kann den Mehraufwand reduzieren, indem er zwei (oder mehrere) parallele Anweisungen zu einer parallelen Anweisung zusammenfasst.

Diese Dissertation beinhaltet das Design, die Implementation und die Evaluation der erweiterten Java Schnittstellen. Die Arbeit beschreibt wie sich potentiell parallele Anweisungen und blockierenden Anweisungen auf die Java Programmiersprache, die JIT Compiler, und das Java Speichermodell auswirken. Zusätzlich wurde eine detaillierte Laufzeitanalyse der automatisch bestimmten Parallelisierungskriterien durchgeführt. Die Resultate zeigen, dass manuell optimierte Parallelisierungskriterien zu einer bis zu 15% schnelleren Ausführung von parallelem Code führen kann.

Wenn die Parallelisierungskriterien allerdings nicht optimiert (oder falsch gesetzt) sind, kann die daraus resultierende parallele Ausführung bis zu 4X langsamer sein.

Acknowledgments

Many people contributed to this thesis. I want to thank my advisor Thomas Gross for providing me with the opportunity to pursue my PhD in his group. I thank Thomas for his guidance through graduate studies and for teaching me the essence of systems research.

I thank the co-examinors Samuel Midkiff and Walter Binder for being on my thesis committee and providing valuable feedback.

I am grateful to the students who contributed to this research and related topics: Jacques Stadler, Christian Kummer, Oliver Kadlcek, and Thomas Anderegg

Thanks to my colleages at the Laboratory for Software Technology. I thank Mathias Payer, my long-term office mate, for numerous inspiring and critical discussions. I also thank all other group members for creating a friendly, productive, and open atmosphere: Zoltan, Michael, Faheem, and Luca. I also want to thank the former members of the Lab for introducing me well to Switzerland and to ETH: Florian, Yang, Susanne, Oliver, Stefanie, Christoph, Niko, and Stefan.

I thank my mother and my family for their support during my graduate studies.

My biggest thanks go to Fabienne for her unconditional support, motivation, and warmth. You and our daughter Annika Luisa make my life complete.

Contents

1

Introduction

1.1 Motivation

Servers, desktop computers, and mobile devices are typically equipped with multi-core processors. To exploit the new hardware resources that are provided by the additional cores, modern programming languages such as Java and C# have built-in support to write parallel applications. However, the support for parallelism is limited to low-level parallel constructs. For example, the Java programming language [40] supports the starting and joining of threads, the definition of synchronized code blocks, and low-level signalling between threads. High-level parallel constructs such as barriers and thread pools are provided as a library. As a result, the interface between the Java programming language and the execution platform of a Java program, the Java Virtual Machine (Java VM), *filters* the high-level information and limits the available knowledge to the low-level primitives. We therefore refer to this interface as a *concurrency-filtering interface* or *concurrency interface*. The concurrency interface defines the knowledge of the concurrency structure of an application (i.e., the assignment of data and code to parallel threads) that is available in a Java binary.

This setup has two implications that can result in performance problems of parallel code. First, due to the filtering of high-level information in the concurrency interface, the semantics of parallel libraries is lost during source code to bytecode compilation. For example, the semantic of a *barrier* that is implemented as an API call cannot be represented in the bytecode. The lost semantics cannot be reconstructed in the Java VM and the just-in-time compiler (JIT compiler) therefore misses optimization opportunities that are highly effective for parallel code [12, 77, 83, 100]. Second, since the Java VM is not in the position to optimize parallel code, the application must manage parallel code optimizations itself. An example of such an application-centric parallel code optimization is the definition of the conditions under which a piece of code is executed in parallel (concurrency conditions). If the concurrency conditions are set poorly, the performance of parallel code can drop way below its potential. It is even possible that a parallel version performs slower than a corresponding sequential version, if the overhead that is associated with every parallel execution (e.g., object allocation(s), inter-thread communication, garbage collection, and scheduling) exceeds the performance gain from a parallel execution.

Figure 1.1 provides an example that shows a generic way to define the concurrency conditions. In Figure 1.1(a), the threshold in line 2 decides if the work is executed sequentially or in parallel. Furthermore, line 5 divides the work into `n` parallel tasks. The variable `n` determines the parallel task granularity. A large number of parallel tasks provides a good load balance. However, the allocation, scheduling, and garbage collection of parallel tasks is associated with a running time overhead (parallel overhead). Finding a number of parallel tasks that provides a good load balance but introduces little parallel overhead is a well-known problem and also referred to as the granularity problem [5]. Figure 1.1(b) illustrates how the `process()` function can be used.

```
1 void process(Work work) {
2  if (work < threshold)
3   do the work sequentially
4  else
5   split work into n parallel tasks
6   invoke n parallel tasks
7 }
```

(a) State-of-the-art parallel code optimization.

```
void caller() {
  ...
  int i = 0;
  for (; i<workItems; i++) {
    process(work);
  }
}
```

(b) Usage example of `process()`.

Figure 1.1: Motivating example.

Figure 1.2 shows the performance of `caller()` for a varying number of work items. For simplicity, assume that the `process()` function generates one parallel task per invocation (`n=1`). The execution times are obtained on a 32-core system. See Chapter 7 for the detailed description of the benchmarking platform.

The x-axis in Figure 1.2 shows the number of work items that can be executed concurrently. The y-axis shows the speedup over a sequential execution. A value of 1 on the y-axis corresponds to the sequential execution time. As illustrated in Figure 1.2, a parallel program can exhibit three behaviors. The behaviors are depicted as region 1, region 2, and region 3. Region 1 illustrates the behavior of a parallel program in which parallelism is underspecified. Since the number of work items is smaller than the number of available cores, the program has a poor load balance. The parallel overhead in region 1 is small, because the number of work items is small as well. In region 2, parallelism is specified well. Region 2 has a good load balance and a low parallel overhead. The maximum speedup in Region 2 is 19; a higher speedup cannot be achieved due to the parallel parallel overhead. In region 3, parallelism is overspecified, since the parallel overhead significantly reduces the performance gain from the parallel execution. For 100,000 work items and more, the execution time of the parallel version is slower than in the sequential version. Programmers aim at writing parallel programs so that the program behavior is similar to the behavior as illustrated in Region 2 of Figure 1.2. However, programmers are always exposed to the risk of underspecifying and overspecifying parallelism.

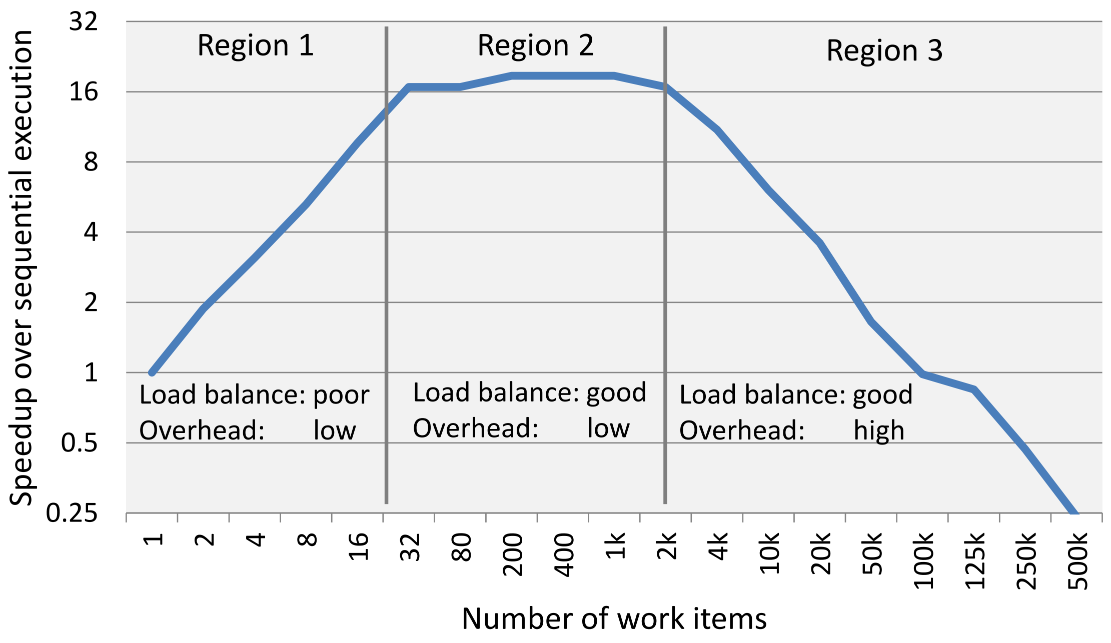


Figure 1.2: Performance impact of parallel code granularity.

The main goal of the thesis is to provide an *automatic* mechanism that manages the execution of parallel programs so that the program has a good load balance and a low parallel overhead. In our model, the programmer is asked to overspecify parallelism. The runtime system dynamically analyses the parallel application and decides what parallel code regions are (i) executed sequentially, (ii) executed in parallel, and (iii) collapsed into one parallel code region. I.e., the system determines the concurrency conditions automatically. These three options allow the runtime to remove the overspecified parallelism. As a result, only the parallel regions that have the behavior as illustrated in region 2 are actually executed in parallel. The main advantage of our approach is that the programmer is not exposed to the risk of overspecifying parallelism. At the same time, the programmer is less likely to underspecify parallelism, since using our approach (in contrast to traditional parallel programming techniques) overspecification does not hurt performance.

This thesis presents the design, implementation, and evaluation of such an automatic mechanism. The mechanism is automated, since the Java VM determines the concurrency conditions and the parallel code granularity based on the knowledge which code regions *may* be executed concurrently and requires no additional input. The proposed mechanism is online, since the concurrency conditions are determined at run-time. As shown in Figure 1.1, traditional parallel programming models require the specification of the concurrency conditions statically in the source code. Finally, the proposed mechanism is feedback-directed, since the concurrency conditions are periodically re-evaluated at run-time.

To implement such a mechanism, the thesis proposes to extend the concurrency-filtering interface between the application and the Java VM so that the *Java VM* can decide how to execute parallel code. We propose to change the standard Java concurrency-filtering interface so that more concurrency information is available to the Java VM. More specifically, *concurrent statements*, *concurrent methods* as well as *synchronization statements* are not filtered by the proposed concurrency interface. Concurrent statements, concurrent methods and synchronization statements are defined in the source language and are also available to the Java VM. A concurrent statement/method *may* be executed in parallel with the statement that follows the concurrent statement or concurrent method call. A synchronization statement enables the synchronization of concurrent statements and concurrent methods. Concurrent statements and concurrent methods have two important properties: First, both constructs allow the specification of fine-grained parallelism, i.e., at the statement and method level, respectively. Second, concurrent statements as well as concurrent methods decouple the specification of parallelism from its execution.

The three new source language constructs map to a Concurrency-aware Intermediate Representation (CIR) in the JIT compiler of a Java VM. CIR is an orthogonal extension to a JIT compiler's sequential intermediate representation so that existing optimizations can be reused without adaptation. To ensure that existing optimizations that operate on CIR generate correctly synchronized code, this thesis present an integration of CIR into the Java Memory Model (JMM) [64]. The thesis is based on the Java platform, since Java programs are executed in a managed environment that is typically equipped with a profiling system and JIT compilers. Furthermore, Java has a memory model that exactly defines the semantics of parallel code. However, the ideas apply to other languages that can be compiled dynamically.

As the proliferation of multi-core systems continues, we see a wider range of parallel systems in use, ranging from modest dual-core systems to systems with 64 cores (or more). Ahead-of-time custom-tuning the software for each platform is impossible, so it is crucial to develop a software infrastructure (VM, JIT compilers, language directives or hints) that allows the automation of this task. Since the Java VM has access to detailed profile information that is only available at run-time, the parallel code granularity is best chosen dynamically.

1.2 Thesis statement

"The decoupling of the declaration of parallelism from its execution as well as a parallelism-aware runtime system enables an efficient execution of fine-grained parallel code."

1.3 Contributions

This theses makes three main contributions:

- **Automatic determination of the concurrency conditions** The thesis presents an algorithm that determines the concurrency conditions fully automatically, at run-time, and based on profile information.
- **Design and implementation of a software platform** The developed software platform provides the integration of concurrent statements, concurrent methods, and synchronization statements into the Java platform. The thesis discusses Java language aspects and modifications to the compilers of the Java platform. Furthermore, this dissertation discusses the integration of the new source language constructs into the JMM [64].
- **Evaluation** The thesis presents the evaluation of the developed software platform and compares the performance against the standard Java platform. In particular, the evaluation presents the performance impact of the extended dynamic profiling system and the extended adaptive optimization system.

1.4 Organization of this thesis

The dissertation is organized as follows: Chapter 2 provides background information that is required to understand the thesis. In particular, this chapter introduces the JMM, discusses compiler optimizations in the JMM, and provides a detailed description of the Java concurrency interfaces. Chapter 3 discusses previous work that is related to this thesis. More specifically, solutions to the granularity problem, concurrency-related compiler optimizations as well as feedback-directed compiler optimizations are discussed. Chapter 4 discusses the foundations for a software platform that can find a good solution to the granularity problem automatically at run-time. The requirements include extensions to the Java source language and extensions to the JIT compiler. Chapter 5 discusses the static and dynamic compilation of the concurrent statements, concurrent methods, and synchronization statements. Furthermore, this chapter presents extensions to the profiling system that enable the Java VM to measure important performance characteristics of parallel code. Chapter 6 presents the toolbox which a JIT compiler can use to change the concurrency conditions. In addition, this chapter presents an algorithm that aims at finding a good solution to the granularity problem. Chapter 7 presents the evaluation of the software platform. Finally, Chapter 8 concludes this thesis.

2 Background information

This chapter contains background material that is required to understand the concepts presented in this thesis. Section 2.1 presents an introduction to the Java Memory Model (JMM). Section 2.2 discusses legal and illegal compiler optimizations in the context of the JMM. Section 2.3 highlights the concurrency interfaces of the Java platform.

2.1 The Java Memory Model

A memory model describes how threads interact through (shared) memory. A parallel Java program consists of multiple threads that can read from and write to shared memory locations concurrently. The JMM defines the set of possible values that a thread can read from a field that is updated by multiple threads. In addition, the JMM aims at bridging the gap between memory models that are easy to understand like sequential consistency [55] but allow a limited set of compiler optimizations, and relaxed memory models that provide more optimization potential. Relaxed consistency models are typically harder to reason about than simple models like sequential consistency. However, several studies have shown that relaxed memory models perform better than strict memory models that do not allow independent instruction reordering [22, 36–38, 72, 94]. For example, loop invariant code motion is potentially legal in a relaxed consistency model but illegal in a sequentially consistent execution.

The JMM is a relaxed consistency memory model and describes reordering constraints of memory operations that the static compiler (source code to bytecode compiler), the just-in-time compiler (JIT compiler) in the Java Virtual Machine (Java VM) (bytecode to machine code compiler), and the hardware (out-of-order execution) must satisfy. Such a memory model is required to guarantee the correctness of code transformations for parallel code, since optimizations that reorder instructions and do not consider parallelism can introduce data races although the code transformation is correct in a sequential context, i.e., the code transformation preserves the intra-thread semantics of the program [23]. If all compilers in the Java platform conform to the JMM such an undesired behavior is impossible.

To bridge the gap between a simple memory model and high quality code, the JMM follows the idea of *data-race-free* models [6, 7]. In these models, correctly synchronized programs are data race free. Two memory accesses represent a *data race* if the two accesses *conflict* and the accesses are not ordered by a happens-before relationship [54]. Two accesses conflict, if the accesses are performed by different threads and both accesses refer to the same shared memory location and at least one of the accesses is a write [82]. To summarize, the JMM provides sequential consistency for correctly synchronized, i.e., data-race-free programs. The next section describes sequential consistency in more detail. What sets the JMM apart from other relaxed consistency models is the fact that the JMM provides guarantees also for programs that contain data races. These guarantees are further discussed in Section 2.1.3.

2.1.1 Sequential consistency

Sequential consistency was defined by Leslie Lamport and requires that "*the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*" [55]. Note that the execution of the instructions of the parallel threads can be interleaved. However, each read must return the value written most recently to the location being read in total order. More formally, each read r of a variable v sees the value written by the write w to v such that:

- w comes before r in the execution order,
- there is no other write w' such that w' comes after w and before r in the execution order.

From the definition above it is easy to see that sequential consistency disallows many compiler optimizations, since reorderings of independent memory accesses break the definition given above.

2.1.2 Terms and definitions of the Java memory model

This section introduces terms and definitions that are used in the JMM.

Shared memory and shared variables: A memory location that can be accessed by more than one thread is called a *shared memory* location or simply *shared memory*. Java objects and arrays are allocated on the Java heap, which is shared memory. *Shared variables* refer to shared memory. All static fields are shared variables. Local variables are stored on the Java stack, which is private to each thread. Consequently, shared variables cannot be stack variables.

Intra-thread actions and intra-thread semantics: An intra-thread action is an action that is not observable by another thread. For example, adding two local variables and storing the result in a local variable is an intra-thread action. Intra-thread semantics are the semantics for single-threaded programs as defined in the Java Language Specification (JLS) [40]. For example, the intra-thread semantics specify data dependencies between local variables. The intra-thread semantics result from the execution of a thread in isolation. If, however, a thread reads a value from the Java heap, the result of the read operation is determined by the JMM.

Inter-thread actions and inter-thread semantics: An inter-thread action is an action that can be detected by another thread. Inter-thread actions are read and writes of shared variables and synchronization actions (see below).

Program order: The program order is a partial order over actions that, for each thread, provides the total order over all actions that are performed by a thread.

Synchronization actions: Synchronization actions are inter-thread actions and include: locks, unlocks, reads of *volatile* variables, writes to volatile variables, actions that start a thread, and actions that detect that a thread is done.

Synchronization order: The synchronization order is the total order over all synchronization actions of an execution. The synchronization order is consistent with the program order and can be represented by *synchronizes-with* edges. The source of a synchronizes-with edge has *release* semantics and the destination has *acquire* semantics. Synchronizes-with edges are defined as follows:

- The unlock action of a monitor m synchronizes-with all lock actions on m that follow the unlock of m according to the synchronization order. In Java, each object has an intrinsic lock. The lock can only be hold by a single thread at a time.
- The write to a volatile variable v synchronizes-with all reads of v that follow the write of v in synchronization order. Volatile variables allow communication between threads without the overhead from mutual exclusion.
- The action that starts a thread synchronizes-with the first action in the started thread.
- The final action in a thread t_1 synchronizes-with the action in another thread t_2 that detects that t_1 has terminated. For example, t_2 can detect that t_1 has terminated by calling the `isAlive()` function on t_1.
- If t_1 interrupts t_2 the interrupt of t_1 synchronizes-with any action by any other thread that detects that t_2 has been interrupted.
- The write of the default value to each variable synchronizes-with the first action in every thread.

- The invocation of a finalizer for an object contains a read of a reference to that object. There is a happens-before edge (see below) from the end of the constructor to that read.

Happens-before: The following notation describes that an action x happens-before an action y: $x \xrightarrow{hb} y$. If the actions x and y are performed by a single thread and x comes before y in program order, then $x \xrightarrow{hb} y$. Furthermore, all synchronizes-with edges induce happens-before edges. The happens-before order is transitively closed. For example, if $x \xrightarrow{hb} y$ and $y \xrightarrow{hb} z$, then $x \xrightarrow{hb} z$.

2.1.3 Happens-before relationships allow undesired behavior

The JMM uses happens-before relationships to define orderings between program actions. For example, the release of a lock of an object o and the acquire of the lock of the same object o are ordered by a happens-before relationship. However, happens-before consistency is not sufficient to provide sequential consistency for (data race free) programs. A program can exhibit out-of-thin-air reads, i.e., a thread can obtain a value from a shared variable v that would have never be written to v in a sequentially consistent execution. Table 2.1 illustrates an example that is happens-before consistent but potentially allows values to appear ”out-of-thin-air”.

Initially, x == y == 0

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`if (r1 != 0)`	`if (r2 != 0)`
`  y = 1;`	`  x = 1;`

Figure 2.1: ”Out-of-thin-air” example.

The code shown in Table 2.1 is correctly synchronized, if executed in a sequentially consistent manner, since neither of the writes will occur. However, the result `r1 = 1` and `r2 = 1` is happens-before consistent, since there is no happens-before relationship that prevents these reads from occurring. Nevertheless such a behavior is considered unacceptable in the JMM.

The JMM defines *causality* which disallows the behavior as described above. The intuition behind causality is that a read returns the value of a write if (i) the write happened before the read, or (ii) the write is already justified. Chapter 17 in the JLS [40] discusses causality formally.

2.2 Legal code transformations in the JMM

This section discusses the reordering of independent memory operations. Table 2.1 summarizes the legal reorderings[1]. All operations are assumed to be accessible by more than one thread and are defined as follows:

Can Reorder	2nd operation		
1st operation	Normal Load Normal Store	Volatile Load `monitorenter`	Volatile Store `monitorexit`
Normal Load Normal Store	**Yes**	**Yes**	No
Volatile Load `monitorenter`	No	No	No
Volatile Store `monitorexit`	**Yes**	No	No

Table 2.1: Legal and illegal instruction reordering optimizations.

- **Normal Loads:** field loads, static variable loads, array load operations.
- **Normal Stores:** field stores, static variable stores, array store operations.
- **Volatile Loads:** volatile field loads, volatile static variable loads.
- **Volatile Stores:** volatile field stores, volatile static variable stores.
- **Monitor Enter:** entry to a synchronized code block.
- **Monitor Exit:** exit from a synchronized code bock.

Reordering normal load and store operations is legal if the reordering retains the inter-thread semantics of the original program. Normal load and store operations can be reordered with volatile loads and `monitorenter` operations. Furthermore, it is legal to reorder normal load/store operations with volatile stores and `monitorexit` operations. I.e., moving normal load/store operations into a synchronized code block is legal. This is also known as the ”roach motel” semantics [64].

All other reorderings shown in Table 2.1 are illegal. Reordering volatile loads and `monitorenter` operations with normal load/stores is illegal. I.e., hoisting normal load/store operations out of a synchronized code block is illegal. In addition, reordering two volatile operations (as well as reordering `monitorenter` and `monitorexit` operations) is always illegal.

[1]http://g.oswego.edu/dl/jmm/cookbook.html

2.3 Concurrency interfaces of the Java platform

This section describes the concurrency interfaces of the Java platform. A Java program typically goes through several transformations before the program can run on a physical machine. Transforming a program from a representation $\mathcal{R}_1$ to another representation $\mathcal{R}_2$ exposes an interface which filters information that cannot be represented in $\mathcal{R}_2$. Thus, $\mathcal{R}_2$ contains less (high-level) information than $\mathcal{R}_1$. If the filtered information is information about the concurrency structure of the application, the interface is referred to as a concurrency-filtering interface. The typical representations of a Java program are: (i) source code, (ii) bytecode, and (iii) machine code. Figure 2.2 illustrates the concurrency interfaces of the Java platform.

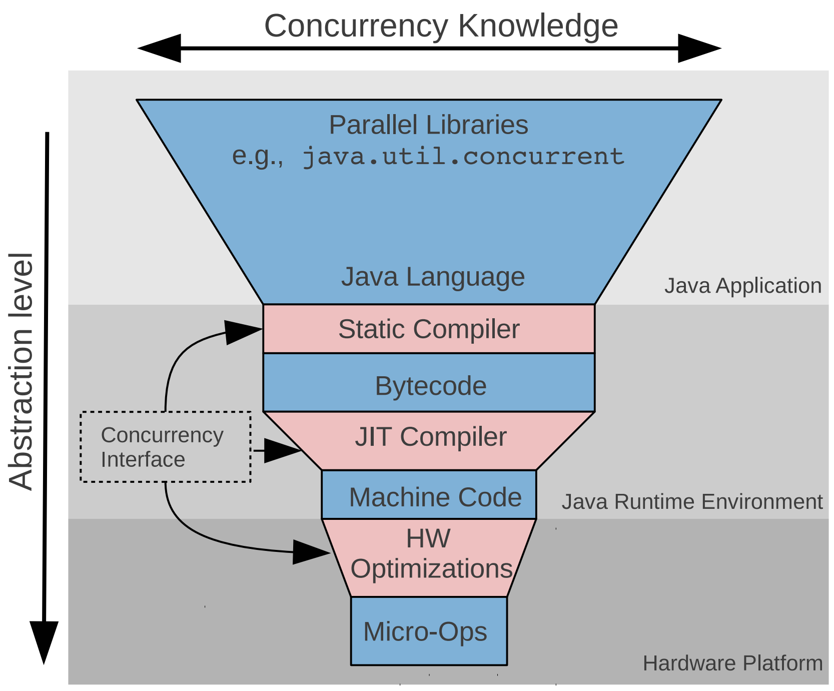


Figure 2.2: Concurrency interfaces in the Java Runtime Environment.

The first concurrency-filtering interface occurs at the translation from Java source code to Java bytecode. This translation is typically done by a static compiler. The static compiler removes high-level information that cannot be represented in the bytecode. For example, the `Executor` interface provides an interface for a thread pool that can execute tasks concurrently. A concrete implementation of a thread pool allows the implementor of the thread pool to specify if and when a task is to be executed by the same thread that puts the parallel task object into the thread pool or by a different thread of the thread pool. Since the `Executor` interface is unknown to the Java VM, this choice is not transparent to the Java VM and the JIT compiler.

The fact that a parallel task can be executed sequentially or in parallel is filtered by the static compiler concurrency interface.

The second concurrency-filtering interface occurs at the translation from Java bytecode to machine code. This translation is done by the JIT compiler in the Java VM, which translates frequently executed methods to executable machine code. Similar to the first concurrency-filtering interface, the JIT compiler removes high-level information that is available in the Java bytecode but cannot be represented in machine code. For example, the Java bytecode contains information about thread creation and thread termination. In Java, a thread is started by calling a particular method that is known to the Java VM and the JIT compiler. This special function call is compiled to a Java VM-internal function call that differs for each Java VM implementation. As a result, by inspecting the code generated by the JIT compiler, the presence of parallelism is, in general, not obvious.

The third concurrency-filtering interface occurs at the hardware-level. For example, modern microprocessors translate machine code to micro-operations at run-time and execute the micro-operations out-of-order. The micro-operations provide micro-architectural details that cannot be represented in machine code.

The remainder of this section discusses the information loss of the concurrency structure of the application at the level of the static compiler and the JIT compiler. Discussing the concurrency interfaces at the hardware level is beyond the scope of this thesis.

2.3.1 Concurrency filter at the static compiler level

This section explains the concurrency-filtering interface of the static compiler with the aid of two different types of parallelism: library-based parallelism and language-based parallelism. Library-based parallelism hides the complexity of writing parallel applications in special libraries. Note that the Java programming language and hence the static compiler is unaware of library-based parallel constructs. As mentioned in the previous section, the `Executor` interface is implemented by special libraries. The Java VM is unaware of these special libraries.

On the other hand, the static compiler and therefore also the Java VM are aware of language-based parallelism (parallel constructs that are defined in the Java language). Although the bytecode represents language-based parallelism, there exist particular properties of the concurrency structure of the application that bytecode cannot express. This section presents an example of such a property.

Library-based parallelism

Many Java programs apply high-level, library-based parallel constructs to abstract the complexity of writing parallel applications. The Java Standard Edition is shipped with the `java.util.concurrent` package that contains a large collection of high-level parallel abstractions such as thread pools and semaphores. Brian Goetz et al. [74] provide guidelines that explain how to apply the high-level parallel constructs that are provided by the `java.util.concurrent` package. Figure 2.3(a) reviews the example from Figure 1.2 and provides a concrete example of how the parallel code optimization strategy[2] can be implemented using a concrete thread pool implementation of the `Executor` interface.

```
 1  void process(int tSize) {
 2    if (tSize < threshold) {
 3      executeSequential();
 4    } else {
 5      int tmp = tSize/2;
 6      Task t = new Task(0,tmp);
 7      threadPool.submit(t);
 8
 9      t = new Task(tmp+1,tSize);
10      threadPool.submit(t);
11    }
12  }
```

(a) Library-based parallel code example.

```
 1  ...
 2  if_icmpge
 3  invokevirtual    //executeSequential
 4  ...
 5  idiv
 6  new              //Task
 7  invokeinterface  //submit
 8  ...
 9  new              //Task
10  invokeinterface  //submit
11  ...
12  return
```

(b) Fragments of the bytecode of Figure 2.3(a)

Figure 2.3: Library-based parallel code optimization.

Recall that the example in Figure 2.3(a) aims at providing an efficient execution by two means: First, if the task size (which correlates to the execution time of a task) is below a certain threshold (line 2) the task is executed sequentially. Second, if the task size is larger than that threshold, the task is executed in parallel. In addition, the code in Figure 2.3(a) aims at providing a good task granularity. In the concrete example, the task is divided into two subtasks that can be executed in parallel (line 5 - line 10).

The hidden agenda behind the threshold is that each parallel execution comes with an overhead in the form of memory allocation, garbage collection, and scheduling. If the overhead from the parallel execution exceeds the potential speedup from using two threads, the parallel version performs slower than the sequential version. In particular, the parallel overhead in Figure 2.3(a) is composed of allocating two parallel task objects in line 6 and line 9 and the scheduling overhead, which consists of waking up the threads in the thread pool as well as moving code and data to the threads in the thread pool. Note that the parallel overhead can vary on different Java VM implementations as well as target platforms (operating system and hardware architecture). Consequently, finding a good value for the threshold is a non-trivial problem [33].

Java bytecode cannot describe the optimization pattern in Figure 2.3(a). As a result, the concurrency interface between the Java source code and bytecode filters this important meta-information. Figure 2.3(b) illustrates fragments of the bytecode

[2]http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

representation of the example shown in Figure 2.3(a). The full bytecode representation is shown in Figure A.1. Task object allocations are compiled to regular Java object creation bytecodes (line 6 and line 9). The submission of task objects to a thread pool is compiled to regular method calls (line 7 and line 10). Since tasks and thread pools are unknown to the Java VM, these method calls are treated like any other method call. To conclude, the JIT compiler is unaware of the parallel execution of the task objects.

Language-based parallelism

The Java programming language [40] defines a small set of low-level parallel primitives which can be used to write parallel code. Figure 2.4(a) shows an implementation of the code in Figure 2.3(a) using Java language primitives. Java abstracts a thread with a regular Java object of type `Thread`. Starting a new thread can only be done by calling the `start()` method on a thread object. The entry point of a new thread into the Java application is the `start()` method, which internally calls the `run()` method of the task object that is passed as a parameter to the constructor of the thread object. Figure 2.4 shows an implementation of the code in Figure 2.3 and the corresponding bytecode representation.

```
1  void process(int tSize) {
2    if (tSize < threashold) {
3      executeSequential();
4    }
5    int tmp = tSize / 2;
6    Task t1 = new Task(0,tmp);
7    Thread thr1 = new Thread(task1);
8    Task t2 = new Task(tmp+1,tSize);
9    Thread thr2 = new Thread(task2);
10
11   thr1.start();
12   thr2.start();
13 }
```

(a) Language-based parallel code optimization.

```
1  ...
2  if_icmpge
3  ...
4  invokevirtual  //executeSequential()
5  ...
6  new            //java/lang/Thread
7  new            //Task
8  new            //java/lang/Thread
9  new            //Task
10 ...
11 invokevirtual  //start()
12 invokevirtual  //start()
13 ...
```

(b) Bytecode of language-based parallel code.

Figure 2.4: Language-based parallel code optimization.

Two task allocations (line 6 and line 8) are required to split the workload into two subtasks. The `Task` class is the template for a task and implements the `Runnable` interface. To facilitate a parallel execution using two threads, two thread objects must be created (line 7 and line 9). These four object creations are typically used to provide a parallel execution using two threads. Similar to high-level parallel constructs, as illustrated in the previous example, using low-level parallel constructs cannot make the semantics of the parallel code optimization pattern transparent to the Java VM. The Java VM and the JIT compiler are unaware of the facts that (i) it is safe the execute the tasks either sequentially or in parallel and (ii) the parallel task granularity can safely be changed at run-time. As a result, the Java VM and the JIT compiler are not in the position to apply the parallel code optimization pattern automatically.

To summarize, the concurrency interface of the static Java compiler filters high-level concurrency information that is contained in parallel libraries. Language-based parallel constructs are not filtered by the concurrency interface and are therefore available to the Java VM. However, the low-level constructs are not sufficient to make the Java VM optimize parallel code automatically.

2.3.2 Concurrency filter at the JIT compiler level

The JIT compiler translates Java bytecode to machine code at run-time. The available information about the concurrency structure that is contained in the bytecode is limited to the language-based parallel constructs. Language-based parallel constructs are compiled to Java VM-internal function calls that implement the corresponding functionality. For example, the Jikes Research Virtual Machine (Jikes RVM) [11] translates the `start()` method call of the `Thread` class to a call to the `start()` method of type `RVMThread`, a Java VM-internal class. The `start()` method of the `RVMThread` class contains a call to the `pthread` API [71] that actually starts the new thread. From inspecting the machine code that is generated by the JIT compiler it is, in general, hard to determine which code regions execute sequentially and which code regions execute in parallel.

Since the Java bytecode contains only limited explicit knowledge about the concurrency structure of the application, there exists only a small set of compiler optimizations that are specifically designed to optimize the execution of parallel code. For example, removing unnecessary synchronization [27] removes redundant `monitorenter` and `monitorexit` bytecodes. The JIT compiler analyses the concurrency structure of the application and identifies a pair of `monitorenter` and `monitorexit` bytecodes as redundant, if the code sequence that should be executed mutually exclusive is guaranteed to be executed by a single thread. If the JIT compiler can show that the object which is used to protect the critical code section does not escape to other threads, the synchronization operation can be removed.

Other compiler optimizations that show good speedups for explicit parallel programming languages like X10 [25] or Cilk [33] cannot be performed by traditional Java JIT compilers, since Java JIT compilers either do not know the concurrency structure of the application, or JIT compilers are not allowed to perform the code transformation due to the rules that are given by the JMM. Consider the example in Figure 2.5, which illustrates the happens-before relationships of Figure 2.4.

Figure 2.5 shows that there is a happens-before relationship between each action in a thread. If one thread starts another thread, starting the thread must happen before the first action in the new thread. Thread inlining potentially adds happens-before relationships. Figure 2.6 illustrates the happens-before relationships when

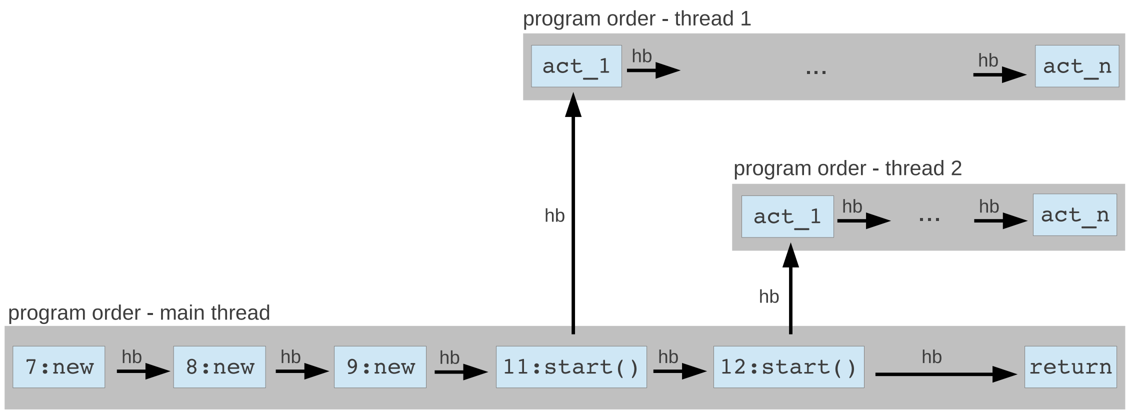


Figure 2.5: Happens-before relationships of actions of Figure 2.4.

the JIT compiler inlines thread 2 into the main thread. In Figure 2.6 inlining thread 2 into the main thread adds a happens-before relationship between `act_n` and the return action from the main thread. Adding this happens-before relationship changes the semantics of the program. For example, if `act_n` in thread 2 waits for a value to be written to main memory and the write happens after the `return` action of the main thread, inlining thread 2 into the main thread results in a deadlock that is introduced by the new happens-before relationship.

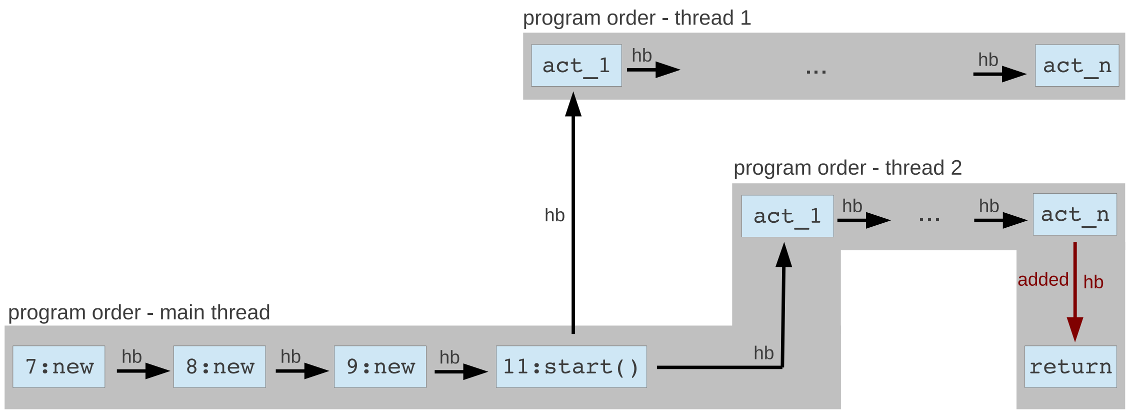


Figure 2.6: Happens-before relationships if Thread 2 is serialized.

To summarize, adding happens-before relationships can introduce new illegal program behaviors [64]. If the JIT compiler performs ”thread inlining” as an optimization that, e.g., serializes the execution of parallel code, the JIT compiler must prove that this code transformation is legal.

3
Related work

This chapter presents three major research directions that are related to this thesis. Section 3.1 discusses the *granularity problem*, which describes the tradeoff between the overhead of fine-grained parallelism and load balancing. Section 3.2 reviews representations of parallel programs in the Intermediate Representation (IR) of modern static and dynamic compilers. Section 3.2 also discusses how compilers optimize parallel code. Finally, Section 3.3 presents feedback-directed optimizations that aim at optimizing the execution of parallel code based on dynamic profiling information.

3.1 The granularity problem

The granularity problem describes the tradeoff between the overhead of fine-grained parallelism and the potential performance gain due to better load balancing. Fine-grained parallelism provides a better load balance compared to coarse-grained parallelism, since there are more tasks that can be distributed to otherwise idle resources. However, fine-grained parallelism incurs more overhead than coarse-grained parallelism, since task creation and scheduling incur a run-time overhead. A good solution to the granularity problem provides a well-balanced system in which the overhead from task creation is kept as small as possible. In general, related approaches that discuss solutions to the granularity granularity problem can be divided into two categories: cost prediction and cost reduction.

3.1.1 Cost prediction

Cost prediction tries to determine when to best cut off parallelism and instead switch to a sequential execution. For example, an efficient recursive parallel version of merge sort dynamically switches to a sequential implementation at a certain recursion depth, i.e., when the subarray that is to be sorted by the task has shrunk to a certain size. If this switch happens too early, the task granularity is too coarse and the system is unbalanced. If the switch happens too late, the task granularity is too fine and the parallel overhead significantly influences the execution time.

Huelsbergen et al. [45] present a framework for dynamic task granularity estimation. The framework has a compile-time and a run-time component. The compile-time component performs an abstract interpretation of potentially-parallel functions and identifies those functions whose complexity (execution time) depends on the size of a data structure. The dynamic component approximates the size of this data structure. The compiler inserts checks to consult the size information based on which the decision is taken whether to start a thread or not. What sets our approach apart from the work by Huelsbergen et al. is that our approach automatically determines whether to execute a task sequentially or in parallel *without* having to identify a data structure that influences the execution time of a task.

An alternative parameter that enables cost prediction is the depth and the height of the recursion tree [73, 96] of a recursive parallel program. However, the depth and the height of the recursion tree are, in general, not the most direct means to predict the execution time of subcomputations, as proposed by Acar et al. [5]. To better estimate the execution time of subcomputations, Acar et al. [5] present oracle scheduling, a semi-automated approach to determine the optimal task granularity. Oracle scheduling requires the programmer to know the asymptotic complexity of the recursive parallel algorithm. Oracle scheduling uses profile information to increase the accuracy of good task granularity. The authors show that oracle scheduling can significantly reduce the task creation overhead compared to the standard Cilk [33] scheduler. In contrast to oracle scheduling, the approach of this thesis performs task granularity optimizations for iterative-parallel programs fully automatically at run-time.

Similar to Oracle scheduling, Lopez et al. [63] propose a solution to the granularity problem, which requires the programmer to provide additional information about the complexity of the recursive parallel computation. The solution is presented in the context of logic programming. One significant difference to [5] is that Lopez et al. do not use profile information to increase the quality of cost prediction. Chen et al. [26] present the design and implementation of a profiler that aims at setting the task granularity such that the tasks show a good last-level cache behavior. The output of the proposed analysis is a grouping of tasks that should be executed by one thread. The group configuration is calculated by using the results of a cache simulator. While the calculated grouping of tasks shows a better cache behavior compared to traditional schedulers, cache simulators are typically too slow to analyze complex large-scale parallel programs. Furthermore, the authors only present the results for a simple parallel merge sort. It remains an open issue to what extent the results are specific towards a particular algorithm, input set, and hardware architecture.

The cost prediction approaches presented in this section are either not fully automatic and require additional input from the programmer, or are designed for a special domain or programming environment. Furthermore, the presented approaches are designed for recursive parallel programs that build up a recursion tree. Other parallel programming models such as non-recursive fork-join models (e.g., OpenMP) cannot be handled by the presented cost prediction models. This thesis presents a solution to the granularity problem for non-recursive parallel programs and requires no manual input.

3.1.2 Parallel overhead reduction

Parallel overhead reduction techniques aim at reducing the cost that is associated with every parallel execution. A lower parallel overhead enables more fine-grained tasks and therefore a better load balance. This section focuses on two techniques that are often used together: work stealing and lazy task creation.

Work stealing

Work stealing is a scheduling technique that aims at providing a good load balance. In a system that supports work stealing, each thread maintains a task queue to which each thread can submit tasks. The task queue is typically implemented as a double-ended queue (dequeue). To balance the load, a thread (the thief) can steal tasks from the task queue of another thread (the victim) if the work queue of the thief is empty. Efficient work stealing aims at minimizing the overhead of maintaining the task queue as well as the overhead from stealing. Both, maintaining the task queue as well as stealing contributes to the parallel overhead.

The idea of work stealing dates back at least to Burton and Sleep [24]. Based on the initial idea by Burton and Sleep numerous researchers have implemented variants of the idea. Existing work stealing schedulers (e.g., Cilk++ and Intel TBB) implement a variant of the ABP (Arora, Blumofe, and Plaxton) algorithm [15, 21]. The Cilk runtime system [33] implements the THE protocol to minimize the overhead from stealing. Blumofe and Lisiecki [20] present the design of Cilk-NOW, which adaptively executes Cilk programs in parallel on a network of workstations.

Blumofe and Leiserson [19] present the first work stealing scheduler for multi-threaded computations with dependences. Arora et al. [15] describe a non-blocking implementation of the work stealing algorithm that is described in [19]. Ding et al. [29] present BWS, a balanced work stealing scheduler for time-sharing multicore systems.

An alternative work stealing algorithm, the A-STEAL algorithm, was recently proposed by Agrawal et al. [10]. The A-STEAL algorithm assumes the existence of a space-sharing OS job scheduler that can allocate arbitrary cores to different applications/tasks. The A-STEAL algorithm incorporates parallelism feedback (e.g., the efficiency of a parallel execution) that determines the number of cores that are allocated to a particular application in the next epoch.

Work stealing is orthogonal to the approach of this thesis. This thesis aims at identifying tasks (among a given set of tasks) that can be executed efficiently in parallel. Work stealing can be used in combination with our approach to, e.g., distribute a given set of tasks to available processing units.

Lazy task creation

Lazy task creation is a technique that aims at reducing the overhead of maintaining the task queue that is used in work stealing schedulers to balance the load. Lazy task creation can successfully reduce the overhead, since tasks are only created and put into the task queue if resources become idle. While lazy task creation can reduce the number of tasks that are generated, the number of steals remains unaffected by the technique. The only means to remove the overhead entirely is to dynamically switch between a sequential version and a parallel version of the function/program. This thesis explores exactly this path.

Mohr et al. [68] present one of the first implementations of lazy task creation. Feeley [31] presents a variant of the lazy tasks presented by Mohr et al. In particular, Feeley proposes to implement the task distribution based on the message passing paradigm. Feeley shows that his implementation of lazy tasks incurs a factor of 2x less overhead compared to the original version.

A prominent example that implements lazy task creation is the Cilk [33] programming language and runtime system. A Cilk program is characterized by two quantities: First, *work* is the time needed to execute the program sequentially. Second, the *critical path length* is the execution time on an infinite number of processors. Cilk is designed around the *work-first principle*: "Minimize the scheduling overhead borne by the world of a computation. Specifically, move overheads out of the work and onto the critical path". Cilk uses a work stealing scheduler that ensures that the overhead from stealing is largely associated with the critical path. However, the victim from which work is stolen incurs an overhead due to mutual exclusion.

Lazy threads [39] describe an approach which allows to execute a parallel call at nearly the efficiency of a sequential call. The main idea is to specialize the representation of a parallel call such that it is ready to be executed by a separate thread (parallel-ready sequential call). Unfortunately, the evaluation of lazy threads is limited to a synthetic benchmark which measures the overhead of executing parallel-ready sequential calls in parallel. The effectiveness of lazy threads for real world applications remains unknown.

Hiraishi et al. [43] present Tascell, a framework that implements lazy task creation using backtracking. Backtracking allows the runtime system to restore a task-spawnable state without incurring the cost of spawning and managing logical threads (like Cilk). Tascell implements backtracking by weaving task information into the runtime stack of a thread instead of explicitly maintaining a task queue. When a victim receives a request from a thief, the victim backtracks through the nested function calls and creates a task for the thief.

Wang et al. [95] present AdaptiveTC, an adaptive task creation strategy for work stealing. AdaptiveTC generates three kinds of tasks: A *task* is a regular task that is pushed onto the task queue. A task can be stolen by a thief. A *fake task* is a plain recursive function and is never put into the task queue. Finally, a *special task* is pushed onto the work queue and indicates the transition from a fake task back to a task. In the AdaptiveTC system, all busy threads avoid creating tasks until a thread becomes idle. AdaptiveTC incurs less overhead on the critical path compared to Tascell and Cilk. At the same time, AdaptiveTC provides a load balance comparable to Cilk and Tascell.

Sanchez et al. [78] aim at reducing the overhead of fine-grained parallelism by a combined hardware-software scheduler. Experimental results show that the hybrid scheduler can significantly increase the scalability of a wide variety of parallel programs. However, the described approach requires special hardware support that is not available on off-the-shelf processors.

While lazy task creation can reduce the overhead of parallelism, there is always an overhead that cannot be removed. As a result, if the task size is sufficiently small, the overhead due to parallelism has a significant performance impact. The approach presented in this theses can *completely remove parallel overhead* by switching to a sequential execution at run-time.

3.2 Compiler optimizations for parallel programs

The second area of related work discusses compiler optimizations for parallel programs. Parallelism can either be specified explicitly by the programmer or compiler analysis can identify parallel code regions automatically. Samuel P. Midkiff provides an overview of fundamental compiler techniques for automatic parallelization [67]. The technique presented in this thesis works for both approaches.

The area is related, since the task granularity is determined by the just-in-time compiler (JIT compiler). As a result, the JIT compiler must be aware of tasks and concurrency. The subsequent two chapters discuss how related work represents parallel code in a compiler and which code optimizations are applied to parallel code.

Unfortunately, many programming languages provide parallelism only as a library. Library-based parallelism implies that parallel constructs are not transparent to the compiler. Consequently, the related work focuses mainly on the optimization of *explicit parallel programs*. In an explicit parallel program, parallel constructs are part of the language and therefore known to the compiler.

3.2.1 Sequential code optimizations and parallel programs

Sequential code optimizations are designed for optimizing the execution time of a piece of code that is executed in a single thread. For example, loop unrolling, common subexpression elimination, reordering of independent memory accesses, and scalar replacement are sequential code optimizations. It is a well-known fact [23, 46] that optimizations that are designed for sequential code cannot be directly applied to parallel programs. Consequently, parallel code must be treated specifically.

Shasha and Snir [82] are one of the pioneers of analyzing parallel programs. The authors present an analysis that guarantees sequential consistency and provides information about conflicting accesses, i.e., shared variables that can be read/written by more than one thread simultaneously. Krishnamurthy and Yelick [52] improve on the complexity of Shasha and Snir's algorithm for cycle detection for programs that are written in single program multiple data (SPMD) style. Chow and Harrison [28] present a compile-time analysis for shared-memory parallel programs. The complexity of the algorithm is exponential. As a result, the algorithm can analyze only programs that have a small number of shared memory locations. Consequently, the algorithm cannot be used in practice.

The papers by Dwyer and Clarke [30], Srinivasan and Wolfe [86], as well as Long and Clarke [62] present analyses that check for specific properties of parallel programs like mutual exclusion, deadlocks, and data races.

The dynamic compiler framework presented in this thesis deals with both issues: optimizing the overheads of parallel programs while retaining all existing, sequential compiler optimizations.

3.2.2 Parallel code optimizations

Parallel code optimizations are designed for optimizing the execution time of parallel code. The compiler must know about parallel constructs to be able to optimize parallel code. There are two possibilities of making a compiler concurrency-aware: First, the compiler analyses sequential code and auto parallelizes sequential code. Second, explicit parallel programming languages provide information about the parallel constructs to the compiler by design.

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. [60] 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 [59]. Sarinivasan et al. present an Static Single Assignment (SSA) form for explicit parallel programs [85].

The main difference of [59, 60, 85] to the concurrency-aware JIT compiler that is presented in this thesis is that making the JIT compiler concurrency-aware requires only minor modifications to the existing infrastructure. For example, the Concurrency-aware Intermediate Representation (CIR) that is presented in Chapter 4 can be transformed to SSA without having to change SSA construction itself. If the reordering of memory load/store instructions happens within a concurrent region or a sequential 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 [60, 85] generate sequentially consistent code [55]. Sequential consistency is a more restrictive memory model than the Java Memory Model (JMM) [64] and therefore forbids many effective optimizations. SC Java code is 10% - 26% slower than Java code compiled according to the JMM [13, 91].

Other examples of parallel program analysis include an optimal bit vector analysis [50] for explicit parallel programs that provides an infrastructure for computing reaching definition, definition-use chains, or analysis for code motion. Grunwald et al. present data flow equations for explicitly parallel programs [41]. Finally, Sarkar presents analysis techniques that are based on a parallel program graph [79]. The papers presented in this paragraph do program analysis in a static compiler. A JIT compiler must generate high quality code in a short time. To the best of our knowledge, this thesis presents the first JIT compiler that operates on an explicit parallel IR (CIR) that can produce high quality code.

There are several OpenMP [4] compilers that perform optimizations in and around parallel regions. For example, Satoh et al. present a reaching definition analysis, memory synchronization analysis and loop data dependence analysis for parallel loops [80]. Tian et al. present multi-entry threading, a compilation technique that can perform compiler optimizations across thread boundaries [93]. Zhang et al. exploit meta information provided by OpenMP directives to analyze barrier synchronization [99].

Cilk [33] extends the C/C++ programming language with parallel calls and a construct to synchronize parallel calls. The semantics of a concurrent statement corresponds to the semantics of the `spawn` keyword in Cilk. The main distinction between the approach of this thesis and Cilk is that concurrent region merging fully removes the parallel overhead of merged concurrent regions. Cilk provides a work-stealing scheduler that shifts a large fraction if the parallel overhead from the sequential version to the parallel execution. However, the sequential execution of a `spawn` statement always incurs a parallel overhead that contributes to the sequential running time.

X10 [25] provides language-based explicit parallelism. Zhao et al. [100] 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. Tardieu et al. present a work stealing scheduler for X10 [92]. The authors in [16] present a load elimination algorithm for X10 programs. Load elimination is a reordering optimization and the algorithm presented in [16] supports the elimination of load across `async` statements. Agarwal [9] et al. present a may-happen in parallel analysis for X10 programs. In another paper, Agawal et al. [8] present an approach for deadlock free scheduling of X10 computations with bounded resources.

Pramod et al. present a technique that can reuse sequential compiler optimizations to parallel code [47] without having to adapt the optimization pass. This work is similar to our work, since existing optimizations can be applied to CIR without any modifications. However, the algorithms that are presented in [47] guarantee sequential consistency, which is a more restrictive memory model than the JMM.

Midkiff and Padua [66] present examples that demonstrate how standard compiler techniques like dead code elimination can produce incorrect code and present an analysis technique that guarantees the correctness of optimizing transformations that are applied to programs with `cobegin/coend` parallelism.

3.3 Feedback-directed optimizations

Feedback-directed optimization systems profile the application at run-time and use the gathered profiling information to adapt the behavior of the application such that a predefined objective is met. For example, one objective can be to optimize for performance or throughput. Another objective can be to use available resources as efficiently as possible or to optimize for power efficiency. Feedback-directed optimizations require online profiling to monitor the behavior of the application. The related work in this section covers monitoring techniques as well as feedback-directed optimizations.

3.3.1 Monitoring techniques

Arnold et al. [14] present the design, implementation, and evaluation of an online feedback-directed optimization system for the Jikes Research Virtual Machine (Jikes RVM). The system is fully automatic, i.e., it requires no offline profiling or additional input from the programmer. The profiles are collected using sampling. The profiling overhead is at most 2%, which is a similar value as shown in Chapter 7. The profile information is used to drive traditional compiler optimizations for sequential code.

Kumar [53] presents a cost/benefit analysis model for determining the hotness of methods to enable eager dynamic compilation. Eager dynamic compilation decreases the amount of time that is spent in the interpreter mode. Interpreting a method is up to 10x slower compared to executing the same method as native code. The analysis computes a cost function based on the bytecodes that are contained in the method. Since the Java bytecode is unaware of parallelism, i.e., there is no bytecode that is associated with starting a new thread, the analysis is parallelism un-aware.

Namjoshi and Kulkarni [70] aim at improving the detection of hot methods by inspecting methods that contain loops of which the loop bound is known prior to entering the loop. If the loop bound is large enough, the method containing the loop is compiled with the optimizing JIT compiler without collecting the profile information that is required by traditional dynamic optimization frameworks. As explained in Chapter 5, this thesis also contains a simple hotness analysis that checks if a concurrent region contains a loop or a call to another method. Straight line code is always compiled to sequential code. However, this thesis does not consider the loop bounds explicitly.

Suganuma et al. [87] present the design of a profiling system for a Java Virtual Machine (Java VM) together with two profile-directed optimizations: method inlining and code specialization. The work is similar to the approach taken in this thesis in that it performs code specialization based on dynamic profile information. The work presented in the thesis specializes concurrent statements or concurrent methods to either sequential or parallel code and groups concurrent statements or concurrent method calls together such that the execution of the grouped concurrent region provides a good load balance and a small parallel overhead.

Schneider et al. [81] present a methodology to make use of hardware performance counters to collect profile information at run-time. Based on the collected profile, the authors adapt a generational garbage collector such that objects are co-allocated. Due to the improved spatial locality, profile-directed object co-allocation results in a reduced L1 cache miss rate by up to 28%. The work by Schneider et al. considers only sequential performance but could be extended to improve the data placement in non-uniform memory architecture (NUMA) systems.

Suganuma et al. [88] present a region-based dynamic compilation technique. Instead of inlining and compiling an entire method, region-based compilation aims at compiling only the code regions that are executed frequently. For example, if one path from the method entry to the method exit is executed frequently, only this part should be compiled and optimized. The other path(s) remain un-compiled. As a result, the compilation time of compiling only hot regions instead of the entire method is shorter. Hot regions are determined using heuristics and dynamic profiles. Region-based dynamic compilation is an interesting technique that can be used to only recompile concurrent regions to sequential or parallel code.

3.3.2 Feedback-directed optimizations for parallel code

Suleman et al. [90] present feedback-directed threading, a framework that uses dynamic profiling information to control the number of threads that are used to execute parallel code. The authors present two optimizations on top of the framework: First, *synchronization-aware threading* predicts the optimal number of threads depending on the amount of data synchronization in the application. Second, *bandwidth-aware threading* predicts the number of threads that are required to saturate the off-chip bus. The authors report an average execution time reduction of 17%. However, the results are obtained from a simulator that simulates an uncommon hardware architecture.

There are several differences to this thesis: First, the online profiling is implemented by dividing the iteration space of a parallel loop into a sampling part and an execution part. The loop iterations in the sampling part contribute only 1% of the total iteration space. If the behavior of the sampling part is not equal to the behavior of the execution part, the methodology does not work. Second, the proposed sampling technique requires extensions to existing hardware. As a result, the approach cannot be used in practice and the reported results are gathered from a hardware simulator. The techniques that are elaborated in this thesis can be implemented on commodity hardware architectures and the reported performance results are gathered from existing hardware systems. Finally, the approach of the authors is limited to parallel loops. Concurrent statements as well as concurrent methods provide the programmer with more opportunities to profit from feedback-directed optimizations.

Raman et al. [75] present DoPE, a framework that aims at separating the concern of exposing parallelism from that of optimizing it. The DoPE runtime system automatically and continuously determines (i) which tasks to execute in parallel (ii) the number of hardware threads that should be used and (iii) the scheduling policy of hardware threads such that locality of communication is maximized. The DoPE system requires the programmer to define objectives that the runtime system aims to achieve. For example, one objective can be to minimize the response time with N threads. One main requirement of DoPE and this thesis is similar, i.e., decoupling the exposure of parallelism from optimization. However, there are two main differences: First, DoPE expects an optimization policy from the programmer while this work aims at a fully-automatic optimization of parallel code. Second, the programming model of this thesis differs from the programming model of DoPE. While DoPE assumes that the programmer specifies "enough" parallelism, the programming model of this thesis asks for an "overspecification" of parallelism. Our runtime system combines the specified parallelism to achieve good performance.

Suleman et al. [89] present a technique that automatically determines the number of cores that should be used to process a stage in a pipelined parallel program. The authors assume that each stage gets at least one core, all cores are allocated, and a core is not shared between threads. As a result, the number of active threads is equal to the number of the available core. Such an assignment of stages to cores leads to a load imbalance if the computational effort of the stages differ significantly. A better load balance can be achieved by over subscription, which the authors do not consider. This thesis takes a fundamentally different approach: this thesis optimizes an oversubscribed system by merging concurrent regions.

Raman et al. [76] present Parcae, a system for flexible parallel execution. The system consists of two parts: the Parcae compiler creates multiple types of parallelism from a parallel code region. The Parcea runtime system comprises a Monitor and an Executor. Parcae enables a platform-wide dynamic tuning of the execution of parallel code. For example, Parcea automatically determines the degree of parallelism, i.e., the number of threads that are used to process a parallel code region. Furthermore, Parcae can dynamically switch between the different parallel version that are generated by the Parcea compiler. Unlike the work that is presented in this thesis, the Parcea system cannot dynamically merge parallel code regions to adapt the parallel code granularity. Another difference compared to this thesis is that Parcea aims at optimizing platform-wide performance. This thesis aims at optimizing code that is executed in the same Java VM instance. The scheduling of multiprogrammed workloads is left to the operating system.

Hall and Martonosi [42] describe a framework for adaptive parallelism in compiler-parallelized code. The runtime system determines the number of threads that are used to execute a parallel loop prior to entering the parallel loop. Similar to Parcae, the runtime system aims at achieving high efficiency instead of optimizing for the peak performance of the application. I.e., if the efficiency drops below a threshold, the resources are assigned to another program that also runs on the system. The work in this thesis considers only a single application and leaves the scheduling of multiple applications to the operating system. However, this thesis also provides the possibility to define an efficiency metric that determines the degree of parallelism of an application.

Parallel Java [48] is a Java-based API that supports both, shared-memory parallel programming and message-based parallel programming in a single unified API. In particular, Parallel Java has the same capabilities as OpenMP [4] and MPI [3]. The OpenMP standard provides a mechanism which allows the OpenMP runtime to dynamically schedule parallel loop iterations to processors. Similar to the work in this thesis, the dynamic loop schedule aims at providing a good load balance in conjunction with low parallel overhead. However, the dynamic loop schedule directive can be applied only to parallel loops whereas the approach taken in this thesis enables the runtime to optimize for a good load balance and low parallel overhead for arbitrary pieces of code. Since Parallel Java is a purely library-based approach, dynamic loop scheduling cannot make use of dynamic profile information that is collected at run-time by the Java VM.

Leiden et al. present the design and evaluation of a task parallel library (TPL) for .NET [61]. TPL describes parallelism as tasks and extends the concept of a task to a *replicable* task. A replicable task can potentially execute the associated action by multiple threads. Similar to this thesis, a task or a replicable task *potentially* execute in parallel, i.e., there are no concurrency guarantees. To provide an efficient parallel execution of tasks, the TPL runtime applies work stealing. Compared to the approach taken in this thesis, TPL cannot change the parallel code granularity as given by the programmer. Furthermore, as TPL is implemented as a library, the underlying execution platform, the Common Language Runtime, is unaware of the parallel structure of the application and is therefore not in the position to optimize the execution of the parallel application.

3.4 Summary

This chapter presented related work in three areas that are related to this thesis. The first area discusses solutions to the granularity problem. The presented solutions can be largely divided into two approaches. The first category tries to estimate the cost of a parallel execution and decides whether to create a new task based on the estimation. The second class of solutions ties to minimize the cost of a parallel execution. This thesis combines both approaches: Online profiling is used to measure the execution time and the overhead of a task. In the next epoch, the system decides whether to generate a new task based on the past behavior. If the behavior of a task changes significantly, the system dynamically adapts to the change. The adaption can trigger a switch from parallel to sequential execution, or vice versa. Alternatively, a set of tasks can be combined to a single task. Such an adaption reduces the parallel overhead and increases the task granularity.

The second area that is related to this thesis is the representation of parallel programs in modern compilers and the set of optimizations that compilers perform on parallel code. Related work considers classical compiler optimizations for parallel programs as well as optimizations that are specifically designed for parallel code. However, most techniques are designed for ahead-of-time compilers, which cannot consider the dynamic behavior of an application. This thesis shows that reducing the overhead of task creation while keeping the system load balanced can be done by a JIT compiler.

Third, feedback-directed optimizations are related to this thesis, since the task granularity is determined at run-time and adapts to the current application behavior. Java VMs choose their code optimizations based on profile information collected at run-time. However, these optimizations are designed to optimize the performance of sequential code and are not designed to reduce the overhead of parallelism. On the other hand, there are numerous papers that discuss how much parallelism should be assigned to an application. The approaches typically do not change the task granularity to reduce the overhead. Instead, they aim at either (1) an optimized efficiency (e.g., power efficiency) or (ii) providing good system-wide balance in multiprogrammed environments.

This thesis is the first to present dynamic task merging, a feedback-directed JIT compiler-assisted technique to find a good solution to the granularity problem.

4 Automatic dynamic task granularity control

This chapter describes the foundations for an automatic dynamic task granularity control system for the Java platform. In the presented system, the Java Virtual Machine (Java VM) is responsible for determining which tasks (i) are executed sequentially, (ii) are executed in parallel, and (iii) are merged. The third option enables the Java VM to change the task granularity as provided in the source code. Task merging results in a coarser task granularity compared to an un-merged version. The system, however, cannot provide a finer task granularity as provided in the source code. Consequently, the presented approach works best with fine-grained tasks that can be merged automatically into a task size that results in an efficient execution. Note that it is not the responsibility of the Java VM to discover tasks that can be executed in parallel.

Automatic dynamic task granularity control requires the separation of concerns of *describing parallelism* in the source code and providing an *efficient execution* of the parallel code. Since this separation of concerns is not provided by the Java platform, this thesis presents an alternative to the existing possibilities of describing parallelism in Java programming language. Section 4.1 presents the source language extensions (concurrent statements, concurrent methods, and synchronization statements) and discusses the implications of the extensions on the static compiler concurrency interface. Section 4.2 describes the extensions to the just-in-time compiler (JIT compiler) concurrency interface. Section 4.3 discusses the integration of the new source language constructs into the Java Memory Model (JMM). Section 4.4 describes the implications of the extended concurrency interface on existing sequential compiler optimizations. Finally, Section 4.5 summarizes the chapter.

4.1 Decoupling the declaration of parallelism from execution

The Java platform inherently ties the declaration of parallelism to its execution. If Java code contains parallel tasks and the code explicitly schedules these tasks to a separate thread for execution, the Java VM must execute the tasks in parallel. I.e., the Java platform provides a concurrency guarantee. Since there are scenarios that require concurrency guarantee (e.g., the model-view-controller design pattern [34]), this thesis introduces concurrent statements and concurrent method that provide *no concurrency guarantee* as *complementary concurrency constructs.* Concurrent statements are potentially executed in parallel with the statements that follow a concurrent statement in program order. Similarly, a concurrent method potentially runs in parallel with the statements that follow the concurrent method call in program order. Concurrent statements, concurrent methods, and synchronization statements require an extension to the static compiler concurrency interface. Concurrent statements and synchronization statements are dynamically compiled to machine code. As a result, the JIT compiler concurrency interface is extended to handle the new source language constructs. Furthermore, this section discusses the differences between the new source language constructs and the Java threading API as well as the interaction with the existing exception handling mechanisms.

4.1.1 Source language extensions

A concurrent statement is a statement that is potentially executed in parallel with the statements that follow the concurrent statement in program order. The Java language defines a *statement* as a complete unit of execution (U_{exec}). For a concurrent statement U_{exec} is potentially parallel. The exact definition of U_{exec} depends on the statement type. The Java programming language defines several types of statements, which are described in Chapter 14 in the Java Language Specification [40]. The subsequent definition describes the exact semantics of three different concurrent statement types.

Definition 1 *(Concurrent statements) Assume a program consists of sequential (normal) statements and concurrent statements. A sequential statement is always executed by the parent thread T_p. A concurrent statement c_i can either be executed by the parent thread T_p or by a separate child thread T_c. If T_p executes c_i, the semantics of c_i correspond to the sequential semantics of c_i. The execution of c_i by T_c depends on the statement type and is defined for the following statement types:*

- Concurrent `if` statement: ***`if`*** *(expression) block_1* ***`else`*** *block_2*
  T_p evaluates the expression; if expression evaluates to true, T_c executes block_1 else T_c executes block_2.

- Concurrent `switch` statement: ***`switch`*** *(expression) SwitchBlock*

  T_p evaluates the expression; T_c executes the SwitchBlock. In particular, T_c executes all statements that follow the matching ***`case`*** *label up to the next* ***`break`*** *label; T_c also executes all statements of the* ***`default`*** *label.*

- Concurrent `for` statement: ***`for`** (init ; expression ; update) block*
  T_p executes the init, expression, and update; T_c executes all instructions of the block.

The Java Language Specification contains more statement types than listed in the previous definition. For example, the `while` statement can be an alternative to the `for` statement in certain scenarios. However, the `while` statement is not well-suited as a concurrent statement, since the evaluation of the expression (see [40] Chapter 14.12) typically depends on the execution of the body of the `while` statement. Since the body of the `while` statement is potentially executed by the child thread (analogous to the definition of a concurrent `for` statement) a concurrent `while` statement introduces a data race between the parent thread and the child thread. Note that the programmer can also introduce a data race between the parent thread and the child thread using concurrent `for` statements. It is the responsibility of the programmer to ensure the concurrent statements are data race free. We leave the identification of further concurrent statements for future work.

Annotating `if` statements, `switch` statements, and `for` statements with the `@Concurrent` annotation marks these statements as concurrent statements. The subsequent definition defines the semantics of concurrent methods.

Definition 2 *(Concurrent methods) Assume a program consists of sequential and concurrent methods. A sequential method is always executed by the parent thread T_p. A concurrent method is either executed by T_p or by a separate child thread T_c. T_p evaluates all values that are passed from the caller of the concurrent method to the concurrent method. T_c executes the method body. A concurrent method can have a return value; the variable that holds the return value must be declared **`final`**. The return value is only valid after a synchronization statement and undefined otherwise.*

A method is declared as concurrent at the method definition. More specifically, annotating a method with the `@Concurrent` annotation defines a method to be a concurrent method.

The last extension to the static compiler concurrency interface is the introduction of synchronization statements. All types of Java statements can be declared as synchronization statements. A synchronization statement is defined as follows:

Definition 3 *(Synchronization statement) The execution of a synchronization statement s_i suspends the current thread prior to executing s_i until all concurrent statements/methods (including nested concurrent statements) that are issued by the calling thread are retired. A concurrent statement/method retires if all effects of all concurrent statements/methods are visible to the calling thread. A synchronization statement proceeds immediately if there are no concurrent statements/methods in flight.*

Synchronization statements can be defined by annotating a statement with `@Sync` in the source code. There is no restriction on which statement types can be synchronization statements.

The current Java Language Specification (JLS) [40] does not support annotations on statements. Concurrent methods can be declared without changing the Java language. Section 5.1 explains how concurrent statements and synchronization statements are implemented in the static compiler.

Figure 4.1 shows a usage example of a concurrent call.

```
1 void caller()
2 {
3   int x;
4   x = foo(1);
5   print(x);
6
7 }
```

(a) Caller.

```
1 int foo(int x)
2 {
3   final int r1 = cCall(x);
4   final int r2 = cCall(x+1);
5   @Sync
6   return r1 + r2;
7 }
```

(b) Synchronization statement.

```
1 @Concurrent
2 int cCall(int x)
3 {
4
5   print(x);
6   return x + 1;
7 }
```

(c) Concurrent method definition.

Figure 4.1: Concurrent call example.

Figure 4.1(a) shows a sequential method `caller()`. Consequently, `caller()` is executed by the parent thread T_p. Method `foo()` shown in Figure 4.1(b) is a sequential method that contains two calls to a concurrent method in line 3 and line 4. In addition, method `foo()` contains a synchronization statement in line 6. Method `foo()` is executed by T_p. Figure 4.1(c) shows the definition of a concurrent method `cCall(int x)`.

The concurrent method can either be executed by T_p or by a new child thread T_c. If T_p calls `foo()` with the parameter $x = 1$ the output of the program can either be "1 2 5" or "2 1 5". Since there is no synchronization statement between the two concurrent method invocations and the execution of a concurrent method is asynchronous, the order in which the concurrent methods are executed is undefined.

The two concurrent calls have a return value each. The variable that stores the return value must be declared `final`. The return value is undefined prior to the execution of a synchronization statement. The return statement in Figure 4.1(b) in line 6 is a synchronization statement. The return values of the two concurrent calls are accessed correctly, since the synchronization of the two concurrent calls happens before the return values are accessed. Consequently, the return value of `foo()` is always 5.

The visibility of local variables and variables that are stored on the Java heap are discussed in more detail later in this chapter.

4.1.2 Interaction with the Java threading API

Concurrent statements/methods and synchronization statements are alternative constructs to describe concurrency in the Java source language. Concurrent statements/methods as well as synchronization statements are orthogonal to and can be used in combination with the standard library-based Java threading API. A fundamental difference to the Java threading API is that concurrent statements/methods are serializable, i.e., the execution of a concurrent statement yields legal program behavior if the concurrent statement is executed by the parent thread as well as if the concurrent statement is executed by the child thread. The serializability enables the Java VM to decide at run-time if a concurrent statement/method is executed sequentially or in parallel (unlike the code in Figure 1.1(a)).

Furthermore, concurrent statements/methods enable the programmer to add concurrency to an existing application without having to define new classes. As a result, the JIT compiler can perform parallel code granularity optimizations at run-time without time-consuming inter-procedural and inter-thread program analysis. Based on the execution time of a concurrent statement/method, the Java VM can switch between sequential or parallel execution. Furthermore, the Java VM can change the parallel code granularity as given by the programmer in the source code. If the Java VM discovers that the currently chosen parallel code granularity results in an inefficient execution of a concurrent statement/method, or a new program input renders the current parallel code setup inefficient, a new parallel code granularity can be obtained by the Adaptive Optimization System (AOS) of the Java VM.

Although the new source language constructs enable the definition of concurrency without having to add code that explicitly generates new objects, a child thread that executes a concurrent statement/method is a regular Java thread. A child thread is represented as an object of type `Thread`. A child thread can acquire and release a monitor, wait on an object, or be notified by another thread. The main advantage of concurrent statements/methods over the Java threading API is that concurrent statements are more flexible. For example, a concurrent call is not bound to a specific type, requires no object allocations, and can have an arbitrary signature i.e., `static` and `private` method calls can be concurrent calls.

4.1.3 Exception handling

Concurrent statements/methods and synchronization statements have an effect on exception handling, since the execution of both concurrent statements/methods and synchronization statements can raise an unchecked exception. Exceptions that are thrown by a concurrent statement that is executed by a child thread are not propagated to the parent thread. However, exceptions can be handled according to the Java specification within the block of statements that are executed by the child thread. An uncaught exception in a child thread stops the execution of the concurrent statement/method. If a concurrent call is defined to throw an exception, the JIT compiler converts the concurrent call to a sequential call.

The parent thread can check if all concurrent statements/methods have retired without throwing an uncaught exception by checking the return value of the `Thread.getError()` function. The return value is valid only if the `Thread.getError()` is called after (in program order) a synchronization statement. The return value shows the number of unsuccessfully completed concurrent statements. Exceptions that are thrown by a synchronization statement are handled according to the Java language specification.

4.2 Extensions to the JIT compiler concurrency interface

This section describes the extensions to the JIT compiler concurrency interface. The extension of the static compiler concurrency interface provides the Java VM with more concurrency information that can be used to optimize the execution of parallel code. Concurrent statements, concurrent methods as well as synchronization statements are represented explicitly in the *Concurrency-aware Intermediate Representation (CIR)* of the JIT compiler. Furthermore, this section presents how parallel constructs of other programming languages and runtime systems can be mapped to CIR.

4.2.1 Synchronization statement

A synchronization statement in the source language maps to a `cSync` CIR-node in the JIT compiler. If a synchronization statement in the source code maps to more than one node in the intermediate representation (IR), `cSync` is inserted *before* the first IR-node that represents the synchronization statement in the source code. For example, the statement $a + b$ maps to more than one bytecode and consequently to more than one node in the IR of a JIT compiler. The `cSync` CIR-node is inserted prior to any IR-node that is responsible for computing $a + b$.

4.2.2 Concurrent statements/methods

A concurrent statement as well as a concurrent method in the source language map to a *concurrent region* in the IR of the JIT compiler. If a concurrent statement has more than one code block that can be executed by the child thread (see Definition 1 in Section 4.1.1), each such code block is an individual concurrent region. For example, each case label of a concurrent switch statement maps to one concurrent region in the CIR. A concurrent region ($\mathcal{C}$) is context-sensitive and is defined by the following tuple: $\mathcal{C} = (\mathcal{S}, \mathcal{B}, \mathcal{I}, \mathcal{P}, \mathcal{M})$.

- $\mathcal{S}$ is signature of the method that contains $\mathcal{C}$. A method can have an arbitrary number of concurrent regions.
- $\mathcal{B}$ is an array of pointers that point to the bytecodes that define the concurrent region. Definition 1 in Section 4.1.1 defines the parts of a concurrent statement that can be executed by the child thread, i.e., the concurrent region.
- $\mathcal{I}$ is a unique identifier for $\mathcal{C}$. $\mathcal{I}$ is necessary since a method can contain multiple concurrent regions.
- $\mathcal{P} \mapsto (p_1, p_2, ..., p_n)$ where $(p_1, p_2, ..., p_n)$ is a list of pointers to the local variable array. Each pointer refers to a local variable that is used as an operand in the bytecode of $\mathcal{C}$. If $\mathcal{C}$ describes a concurrent method invocation statement, $(p_1, p_2, ..., p_{n-1})$ point to the parameters of the concurrent call (operand stack). p_n points to the return value of the concurrent call. The JIT compiler uses the pointers to expand the concurrent region to multi-threaded code.
- $\mathcal{M} \mapsto (m_1, m_2, ..., m_n)$ where $(m_1, m_2, ..., m_n)$ is a list pointers into the bytecode array. Each pointer refers to a bytecode that represents a synchronization statement that matches $\mathcal{C}$. A synchronization statement matches a concurrent region if there exists a path (in the control flow graph) from the concurrent region to m.

For example, the concurrent call in Figure 4.1(b) in line 3 maps to the following concurrent region:

$$\begin{aligned}\mathcal{C} = (\ &\mathcal{S} : \texttt{foo:(I)I},\\ &\mathcal{B} : 2 \rightarrow \texttt{invokevirtual},\\ &\mathcal{I} : 0,\\ &\mathcal{P} : (0,\ 1,\ 2),\\ &\mathcal{M} : 12 \rightarrow \texttt{iload\_2}\ )\end{aligned}$$

The signature $\mathcal{S}$ of $\mathcal{C}$ is `foo:(I)I` because the concurrent method is defined in function `foo()` and has one integer parameter and returns an integer. The pointer array $\mathcal{B}$ contains one entry, the value 2. The value of 2 points to the `invokevirtual` bytecode that issues the concurrent call `cCall()`. The unique identifier $\mathcal{I}$ is 0, since the concurrent call in line 3 of Figure 4.1(b) is the first concurrent statement/method that is encountered. $\mathcal{P}$ points to the local variable array at index 0, 1, and 2. Index 0 points to the `this` pointer. Index 1 points to the parameter that is passed to `cCall()`. Index 2 points to the return value of `cCall()`. Finally, $\mathcal{M}$ contains a single entry (12), which points to the `iload_2` bytecode. `iload_2` is the first bytecode that represents the synchronization statement that matches $\mathcal{C}$. Note that the synchronization statement in the source code (`return r1 + r2`) is represented by the following sequence of bytecodes: 12→`iload_2`, 13→`iload_3`, 14→`iadd`, 15→`ireturn`.

The JIT compiler uses the information provided by a concurrent region to generate the CIR. CIR explicitly represents a concurrent region in the IR of the optimizing JIT compiler using four new CIR nodes:

- `cStart`: marks the beginning of a concurrent region. Instructions that follow the `cStart` CIR-node[1] are part of the same concurrent region. All instructions that reside in a concurrent region are executed by the same thread (either by the parent thread or by a new child thread). The `cStart` CIR-node must dominate all IR nodes of the concurrent region.
- `cEnd`: represents the end of a concurrent region. This means that the IR-node that follows the `cEnd` CIR-node in program order is not part of the concurrent region.
- `cBarrier`: represents a boundary across which the JIT compiler is not allowed to reorder instructions. The `cBarrier` CIR node guarantees sequential consistency for data-race free programs.
- `cSync`: blocks until all issued concurrent statements have retired. Section 4.3.3 presents more details on the JMM semantics of the `cSync` CIR-node.

Figure 4.2(b) shows the CIR of method `foo()` in Figure 4.2(a).

```
 1  int foo(int x) {
 2
 3    final int r1 = cCall(x);
 4
 5
 6
 7
 8
 9    final int r2 = cCall(x+1);
10
11
12
13    @Sync
14    return r1 + r2;
15  }
```

(a) Example with two concurrent calls and one synchronization statement.

```
 1  cStart(concurrent region 0)
 2  cBarrier
 3  r1 = call cCall(x)
 4  cBarrier
 5  cEnd(concurrent region 0)
 6  t1 = x + 1;
 7  cStart(concurrent region 1)
 8  cBarrier
 9  r2 = call cCall(t1)
10  cBarrier
11  cEnd(concurrent region 1)
12  cSync (synchronization statement)
13  cBarrier
14  return r1 + r2
15  cBarrier
```

(b) Concurrency-Aware Intermediate Representation (CIR) of Figure 4.2(a)

Figure 4.2: Representation of concurrent statements and synchronization statements in the source language and the IR of the JIT compiler.

Each call to `cCall` is contained in a separate concurrent region. Concurrent region 0 ($\mathcal{C}_0$) ranges from line 1 - line 5 and concurrent region 1 ($\mathcal{C}_1$) ranges from line

[1]We use the term instruction (or IR node) in the Java VM context, because one statement in the source language can map to multiple instructions in the IR of the JIT compiler.

7 - line 11. All instructions that are contained in a concurrent region are enclosed between two `cBarrier` CIR-nodes as well as a `cStart` and a `cEnd`-CIR-node. The instructions that represent the synchronization statement in the source code are enclosed between two `cBarrier` CIR-nodes. The `cBarrier` CIR-node in line 15 is redundant since, since there is no instruction that follows the `cBarrier` CIR-node.

The JIT compiler compiles concurrent statements/method calls to regular IR nodes. For example, a concurrent call is translated to a standard Jikes Research Virtual Machine (Jikes RVM) IR node. The explicit representation of concurrency in the IR allows the JIT compiler to inline the body of a concurrent method into the body of the caller. The inlined code is then contained in a single concurrent region. Inlining concurrent calls extends the scope of the JIT compiler so that optimizations can be performed across thread boundaries. See Section 6.1.3 for a discussion of the optimizations.

4.2.3 Other front-ends for concurrent regions

A concurrent region in the JIT compiler concurrency interface can be seen as a back-end to which a front-end, e.g., a concurrent statement can be mapped. This section presents examples of other parallel patterns and programming language constructs that can be mapped to concurrent regions. In particular, this section discusses the producer-consumer pattern [34], which can be implemented using concurrent statements/methods. Furthermore, any kind of fork-join parallelism that is present in OpenMP-like [4] programs can easily be mapped to concurrent regions. Finally, algorithms that have divide and conquer parallelism like typical Cilk [33] programs can also be represented with concurrent regions.

Producer-consumer pattern

The producer-consumer pattern aims at distributing tasks to execution units and provides, in general, M producers and N consumers. The producers generate tasks and put the tasks into a shared work queue. The consumers access the shared work queue and fetch available tasks to process the tasks concurrently. Web servers often apply the producer-consumer pattern: A producer takes incoming requests and the consumers process these requests in parallel. The producer-consumer pattern is exposed to the granularity problem, since generating and scheduling a task introduces a run-time overhead.

Concurrent statements/methods implicitly provide producer-consumer semantics. The parent thread can be seen as the producer and a child threads can be seen as a consumers. Recall that the semantics of a concurrent region imply that the parent thread as well as the child thread can process the concurrent region and that there is no ordering guarantee (unless provided by a synchronization statement) with respect to the order in which individual parallel tasks are processed.

Fork-join model

Many parallel programming languages and systems like OpenMP and X10 apply the fork-join model. Java provides fork-join parallelism at a low level (i.e., starting and joining threads) but also at a high level of abstraction via the fork-join framework that is part of the `java.util.concurrent` package. The fork-join model requires two operations: The fork operation generates a set of worker threads that execute a parallel code region. The join operation waits for all worker threads to complete executing the parallel code region. Examples of parallel regions in an OpenMP program are: parallel loops, parallel code sections, and parallel tasks.

Fork-join parallelism can be easily compiled to concurrent regions: The fork operation corresponds to the semantics of the `cStart` CIR-node and the join operation corresponds to the semantics of the `cSync` CIR-node. The code between the `cStart` CIR-node and the `cEnd` CIR-node defines the parallel code region, which the worker threads can execute in parallel.

Divide and conquer pattern

Divide and conquer algorithms recursively break up a problem into subproblems of smaller size until the subproblem can be solved directly. The solutions of the subproblems are combined to the overall solution. A parallel divide and conquer algorithm solves the subproblems in parallel and synchronizes on the combination of the subsolutions. Recursive parallel divide and conquer algorithms are exposed to the granularity problem, since the division into a subproblem incurs a parallel overhead that must be compensated by the speedup of a parallel execution to achieve a performance gain. For example, a parallel merge sort implementation can spawn a separate task for each subarray and synchronize on the merging of the subtasks. To provide an efficient parallel implementation, the programmer defines a minimum size of the subarray and if the size of a subarray is smaller than this minimum size, the subarray is solved sequentially.

Since concurrent regions can be nested (a concurrent region can contain a concurrent region), recursive divide and conquer algorithms can be expressed using concurrent regions.

4.3 JMM semantics of the JIT compiler concurrency interface

The JMM [64] defines the semantics of parallel Java code. The JMM is a relaxed-consistency model that allows the JIT compiler to reorder independent memory accesses. Despite the possible memory reordering, the JMM guarantees sequential consistency for correctly synchronized programs, i.e., programs that do not contain a data race. If a program is not correctly synchronized the instruction reorderings can result in surprising program behaviors. Consequently, the JMM gives strict rules that define precisely which values can be obtained by a read of a shared field that is updated by multiple threads.

Concurrent statements, concurrent methods, and synchronization statements can write to shared memory and are explicitly defined and synchronized by the four CIR-nodes (`cStart`, `cEnd`, `cBarrier`, `cSync`). Therefore, the CIR-nodes must be integrated into the JMM, which is described in this section.

4.3.1 JMM semantics of `cStart` and `cEnd`

A concurrent region begins with the `cStart` CIR-node. `cStart` is an inter-thread action, since the statements that are contained in the concurrent region can be executed in parallel with statements that follow the concurrent region in program order. As such, `cStart` semantically starts a new thread that executes the concurrent region. More precisely, `cStart` either starts a new thread if the concurrent region is executed by the child thread, or not if the concurrent region is executed by the parent thread. To be on the safe side, the JIT compiler conservatively assumes that `cStart` always starts a new thread.

Since starting a thread is an inter-thread action in the JMM, the `cStart` CIR-node is an inter-thread action as well. In particular, `cStart` is added to the existing set of synchronization actions. Starting a 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 (i.e., field values that are cached in registers) of the parent thread is transferred to the child thread. `cStart` has acquire semantics for the child thread, because the child thread must update its 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 that the child thread performs in the 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`. The child thread commits the local view of memory to global memory. Consequently, `cEnd` has release semantics for the child thread. Concurrency invariant instructions (see Section 4.4) can be reordered across `cStart` and `cEnd` nodes. Such reorderings allow the JIT compiler to, e.g., adapt the parallel code granularity.

4.3.2 JMM semantics of `cBarrier`

The `cBarrier` CIR-node is a barrier for the JIT compiler across which no instruction reordering is allowed. The `cBarrier`-CIR-node has neither acquire nor release semantics. As discussed in Chapter 6, 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 (see Section 4.4) apply. 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 [64]:

Initially, x == y == 0

foo()
0: `cCall1();`
1: `cCall2();`
2: `cCall3();`

(a) Caller.

cCall1()	cCall2()	cCall3()
1: `r1 = x;`	4: `r2 = x;`	6: `r3 = y;`
2: `if (r1 == 0)`	5: `y = r2;`	7: `x = r3;`
3: `  x = 1;`		

(b) Callees.

Figure 4.3: ”Bait-and-switch” example. r1 == r2 == r3 == 1 is not legal.

The code in Figure 4.3(a) contains three consecutive concurrent calls. The method bodies of the three concurrent calls `cCall1`, `cCall2`, and `cCall3` are illustrated in Figure 4.3(b). `x` and `y` are fields that are potentially 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`, if `1` is assigned to the field `x`. The behavior in question requires ”out-of-thin-air” reads, which violate the security guarantees that are provided by Java and are not legal in the JMM. Figure 4.4 shows the effects of a JIT compiler optimization that introduces an out-of-thin-air read and is legal *without* the `cBarrier` node. Note that the code shown in Figure 4.4 is a variant of the code shown in Figure 4.3 that allows `r1 == r2 == r3 == 1`.

Figure 4.4(a) shows the code in which the JIT compiler inlined the calls `cCall1` and `cCall2` into function `foo()`. Concurrent call inlining expands the scope of the JIT compiler across thread boundaries. 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 illustrated in Figure 4.4(b), the behavior in question (`r1 == r2 == r3`) is possible if the following sequence of statements is executed: 1, 2, 3, 11, 12, 4-10. The `cBarrier` CIR-node prohibits this illegal behavior.

4.3.3 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 the `join()` function on every child thread. The join operation ensures that all children have finished execution and updated their local view of

Initially, x == y == 0

cCall1() and cCall2()	cCall3()
0: `cStart;`	10: `r3 = y;`
1: `cBarrier;`	11: `x = r3;`
2: `r1 = x;`	
3: `if (r1 == 0)`	
4: `x = 1;`	
5: `cBarrier`	
6: `r2 = x;`	
7: `y = r2;`	
8: `cBarrier;`	
9: `cEnd;`	

(a) Effects of "thread inlining".

Initially, x == y == 0

cCall1() and cCall2()	cCall3()
0: `cStart;`	10: `r3 = y;`
1: `cBarrier;`	11: `x = r3;`
2: `y = 1;`	
3: `r1 = x;`	
4: `if (r1 == 0)`	
5: `x = 1;`	
6: `cBarrier`	
7: `r2 = 1;`	
8: `cBarrier;`	
9: `cEnd;`	

(b) Code optimizations across concurrent regions.

Figure 4.4: Effects of concurrent call inlining. Must not allow: r1 == r2 == r3 == 1.

memory to global memory. The parent thread synchronizes the local memory with global memory. Since `join()` is a synchronization action in the JMM, `cSync` is a synchronization action.

4.3.4 Summary of JMM semantics of the CIR nodes

Table 4.1 summarizes the JMM semantics of the four CIR nodes.

CIR-node	parent thread	child thread	reorder with IR node
cStart	release	acquire	yes: concurrency-inv
cEnd	-	release	yes: concurrency-inv
cSync	acquire	-	no
cBarrier	-	-	no

Table 4.1: Java Memory Model semantics of CIR nodes.

Table 4.1 shows the JMM semantics for the parent thread and the child thread for each CIR node. In addition, Column 4 shows for each CIR node if there exists an IR node with which it can be potentially reordered. Such a reordering is only legal of the IR node is concurrency invariant. Concurrency invariance is defined in the next section.

4.4 Compiler optimizations in CIR

This section discusses the interoperability of the CIR with existing compiler transformations. Existing code transformations can be classified into optimizing intra-thread actions and optimizing inter-thread actions. Furthermore, an optimization can involve the reordering of independent instructions. Reordering of independent instructions is an important optimization that allows the JIT compiler to change the order in which the statements appear in the source code. For example, loop invariant code motion is a reordering optimization which can involve the reordering of intra- and inter-thread actions. A reordering optimization in CIR can be further classified into an intra-region reordering optimization and an inter-region reordering optimization. Figure 4.5 illustrates intra-region and inter-region reorderings.

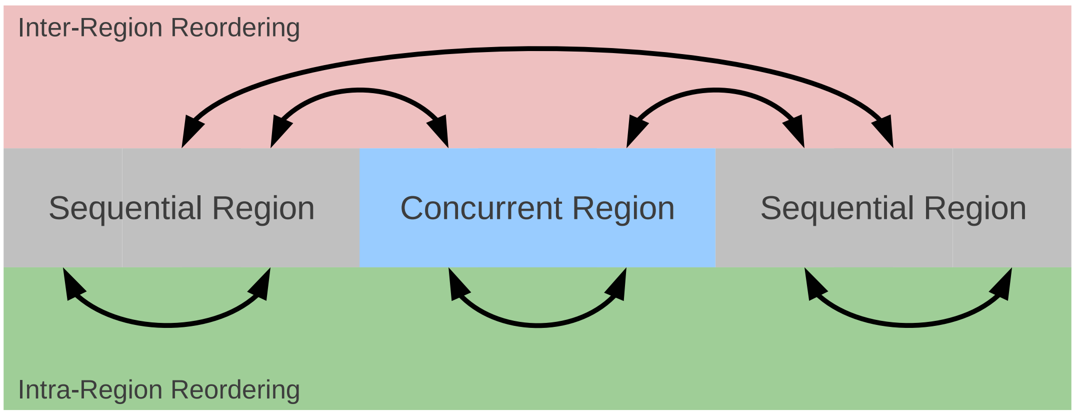


Figure 4.5: Intra- and inter-region reordering in CIR.

Figure 4.5 shows two sequential regions and one concurrent region. An intra-region reordering is a reordering of two instructions that are contained in the same sequential or concurrent region. Intra-region reordering optimizations are legal if they comply to the rules that are given by the Java language semantics (for reordering intra-thread actions) and the JMM for inter-thread actions.

Table 4.1 illustrates which documents specify whether a reordering code transformation is legal.

reordering	**intra-thread action**	**inter-thread action**
intra-region	Java Language Semantics	JMM Semantics
inter-region	JMM Semantics (including CIR nodes)	JMM Semantics (including CIR Nodes)

Table 4.2: Java Memory Model semantics of CIR nodes.

The correctness of intra-region reordering code transformation for intra-thread actions is defined in the Java Language Specification JLS. The correctness of reordering code transformations of inter-thread actions is defined in the JMM. The correctness of inter-region reordering code transformations for intra-thread actions as well as inter-thread actions is defined in the JMM that is augmented with the semantics of the four CIR-nodes. The remainder of this section specifies the conditions under which inter-region reordering optimizations are legal. Such a reordering is called a *concurrency invariant* reordering and produces sequentially consistent code for data race free programs.

Definition 4 defines concurrency invariance. The remainder of this thesis uses the following notation to describe a concurrency invariant instruction.

Definition 4 ***(Concurrency Invariance)*** *An instruction I that is defined in a home region $\mathcal{C}_H$ and is concurrency invariant to a set of target regions $\{\mathcal{C}_{T1}, ..., \mathcal{C}_{Tn}\}$ is defined as: $I \xrightarrow{C_{inv}} \{\mathcal{C}_{T1}, ..., \mathcal{C}_{Tn}\}$. If I is concurrency invariant to $\mathcal{C}_T$, I can either be moved into $\mathcal{C}_T$ by reordering i with* ***`cStart`*** *(if I is defined before (in program order) $\mathcal{C}_T$), or by reordering I with* ***`cEnd`*** *(if I is defined after (in program order) $\mathcal{C}_T$). An instruction I is concurrency invariant if and only if:*

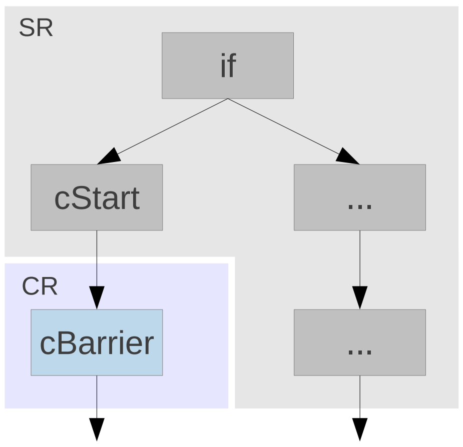
- *I cannot throw an exception;*
- *I does not change the control flow;*
- *moving I from $\mathcal{C}_H$ to $\mathcal{C}_T$ does not change the intra-thread semantics of $\mathcal{C}_H$ and $\mathcal{C}_T$;*
- *moving I from $\mathcal{C}_H$ to $\mathcal{C}_T$ does not change the inter-thread semantics of $\mathcal{C}_H$ and $\mathcal{C}_T$.*

Exceptions

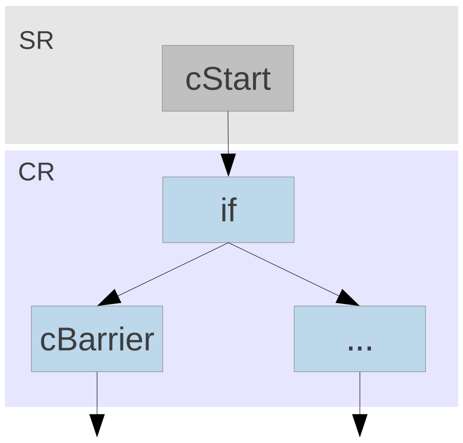
A concurrency invariant instruction cannot be a potential exception-throwing instruction (PEI), since exceptions are not propagated from the child thread to the parent thread. For example, if a PEI throws an exception that is caught in the parent thread but not in the child thread, the uncaught exception causes the concurrent region to finish unsuccessfully. Such a code transformation introduces a new behavior and is therefore an illegal compiler transformation. As a result, all instructions that allocate memory are not concurrency invariant, since memory allocation can raise an `OutOfMemoryError` exception. Furthermore, instructions that access data from a different object than the current object (which is accessed by the `this` pointer) are not concurrency invariant, since accessing a different object can throw a `NullPointerException`. However, compiler analysis [49] can convert a PEI into an instruction that cannot throw an exception.

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 4.6, which depicts the effects of reordering the `cStart`-CIR-node with an `if-node`. Figure 4.6(a) shows a sequential region (SR) that conditionally executes a concurrent region ($\mathcal{C}$). The `then` part starts the concurrent region and the `else` part continues executing a sequential region. Figure 4.6(b) shows the code when the JIT compiler moves the `if`-node into the concurrent region. In Figure 4.6(b) the child thread executes the `then` part as well as the `else` part of the if-statement. Such a behavior is illegal since it changes the intra-thread and inter-thread semantics of the program.



(a) Reordering branch instructions.



(b) Branch instruction reordering

Figure 4.6: Effects of reordering branch instructions.

Intra-thread and inter-thread semantics

The following two sections discuss the conditions under which an inter-region reordering does not change the intra-thread semantics and the inter-thread semantics of the child thread and the parent thread.

4.4.1 Reordering intra-thread actions

Intra-thread actions are actions that are not visible to another thread. For example, the addition of two local variables is an intra-thread action since local variables are maintained on the stack. The stack is private to each thread. In the JMM 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 local variables that are alive [69] and used in a concurrent region to the values of the corresponding variable in the parent thread.

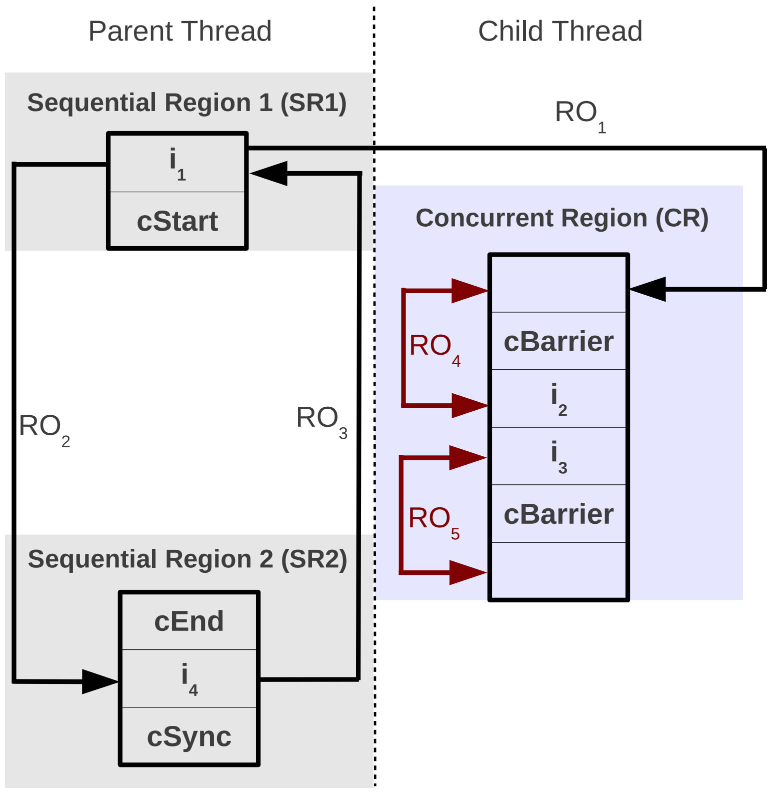


Figure 4.7: Reordering constraints for standard compiler optimizations in CIR.

Figure 4.7 shows two sequential regions and one concurrent region. Furthermore, Figure 4.7 illustrates potentially concurrency invariant reorderings (RO_1, RO_2, and RO_3) as well as reorderings that are always illegal (RO_4 and RO_5). The subsequent paragraph defines the conditions under which RO_1, RO_2, and RO_3 are legal.

Concurrency invariant intra-thread actions

Assume that i_1 defines a local variable l_1 and i_4 defines a local variable l_2. Furthermore, assume that i_1 and i_2 are intra-thread actions that do not change the control flow and cannot throw an exception.

RO_1 : $i_1 \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l_1)$ is not used in $\mathcal{SR}2$. Note that changes to local variables that are performed by the child thread are not visible to the parent thread.

RO_2 : $i_1 \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l_2)$ is not used in $\mathcal{C}$. Killing a live $def(l_1)$ changes the intra-thread semantics of the child thread.

RO_3 : $i_4 \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l_2)$ does not kill another $def(l_2)$ in $\mathcal{C}$. Killing a live $def(l_2)$ changes the intra-thread semantics of the child thread.

RO_4 and RO_5 are, in general, illegal for intra-thread actions (see Section 8 in [64]). The JIT compiler can use standard liveness analysis [69] to determine the conditions listed above.

4.4.2 Reordering inter-thread actions

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

Normal loads and normal stores

Assume that i_1 and i_4 do not change the control flow and cannot throw an exception. Furthermore, assume that a load instruction (ld) loads the value from $o.f$ into a local variable l and a store instruction (st) stores the value of v to the field $o.f$. The reordering RO_1, RO_2, and RO_3 from Figure 4.7 are concurrency invariant if the following conditions hold.

$RO_1 : ld$ $l = ld(o.f) \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l)$ is not used in $\mathcal{SR}2$ and there is no $st(o.f, v)$ until the next `cSync` CIR-node. For example, if $i_4 = st(o.f, v)$ RO_1 introduces a data race, because the two instructions $l = ld(o.f)$ and $st(o.f, v)$ are performed by different threads.

$RO_1 : st$ $st(o.f, v) \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if there is no $ld(o.f)$ until the next matching `cSync` CIR-node. For example, if $i_4 = ld(o.f)$ then RO_1 introduces a data race.

$RO_2 : ld$ $l = ld(o.f) \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l)$ is not used in $\mathcal{C}$.

$RO_2 : st$ $st(o.f, v) \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if there is no $ld(o.f)$ in $\mathcal{C}$. Note that if $i_1 = st(o.f, v)$ and $i_2 = ld(o.f)$, i_2 must see the update that is performed by i_1.

$RO_3 : ld$ $l = ld(o.f) \xrightarrow{C_{inv}} \{\mathcal{C}\}$ if $def(l)$ is no used in $\mathcal{C}$.

$RO_3 : st$ $st(o.f, v) \xrightarrow{C_{inv}} \{\mathcal{C}\}$. The store of i_1 and possible $l = ld(o.f)$ in $\mathcal{C}$ are not ordered by a happens-before relationship. This is a data race.

The JIT compiler uses traditional compiler analysis to determine the conditions listed above. If the JIT compiler cannot determine the conditions, the JIT compiler conservatively assumes that the instruction is not concurrency invariant. RO_4 and RO_5 are not legal, since these reordering can introduce new program behaviors (see Section 8 in [64]).

Volatile loads and volatile stores

A volatile load operation is not concurrency invariant. The reason is that the ordering guarantees that are given by volatile semantics are broken. Consider the example as shown in Figure 4.8. Figure 4.8(a) shows the original version and Figure 4.8(b) shows a variant of Figure 4.8(a) in which the volatile load of y of Thread 2 is moved into the concurrent region.

Initially, x == y == 0;
y is declared `volatile`

Thread 1	Thread 2
1: `x = 1;`	3: `r1 = y + 2;`
2: `y = 1;`	4: `cStart;cBarrier`
	5: `  r2 = r1 * 2;`
	6: `cBarrier;cEnd;`
	7: `r3 = x;`

(a) r1 == 3, r3 == 0; disallowed by JMM.

Initially, x == y == 0;
y is declared `volatile`

Thread 1	Thread 2
0: `x = 1;`	2: `cStart;cBarrier;`
1: `y = 1;`	3: `  r1 = y + 2;`
	4: `  r2 = r1 * 2;`
	5: `cBarrier;cEnd;`
	6: `r3 = x;`

(b) Allows r1==3 and r3==0;

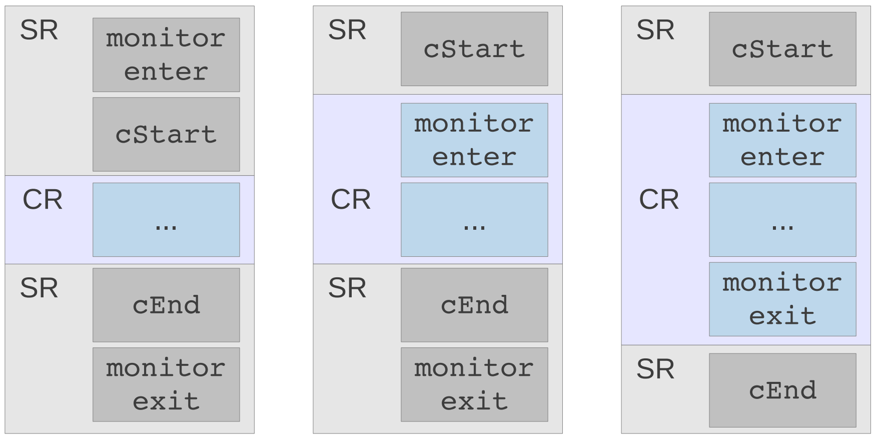
Figure 4.8: Reordering volatile load with `cStart`.

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 Operation 2 and as a result also Operation 1. However, Figure 4.8(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, Operation 6 in Figure 4.8(b) can be executed prior to Operation 3. Such a behavior is illegal in the original code. It is easy to see that reordering a volatile load with the `cEnd` CIR-node (e.g., Operation 7 in Figure 4.8(a)) causes an equivalent behavior for a concurrent region that follows Operation 7.

Moving a volatile store into a concurrent region introduces, in general, a new program behavior that is not possible in the original program and is therefore an illegal operation. Similar to the example shown in Figure 4.8, 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 that the code in the child thread is executed asynchronously with the code in the parent thread that follows the concurrent region.

monitorenter and monitorexit

The `monitorenter` and `monitorexit` bytecodes implement the `synchronized` statement. `monitorenter` attempts to gain the ownership of the intrinsic lock that is associated with each Java class and object. Java locks are reentrant, i.e., the same thread can acquire a lock multiple times. Each time a thread acquires the lock of an object, a counter that is associated with each lock is incremented by 1. The `monitorexit` bytecode decrements the lock count. If the lock count reaches 0, the thread releases the monitor. Moving a `monitorenter` and/or a `monitorexit` operation into a concurrent region is an illegal code transformation. Figure 4.9 shows an example of such an illegal reordering.



(a) Original version. (b) Monitorenter reordering. (c) Monitorexit reordering.

Figure 4.9: Effects of reordering the monitor operations.

Figure 4.9(a) shows the original version, in which the `monitorenter` operation is ordered before the concurrent region. The corresponding `monitorexit` operation is ordered after the concurrent region. There are two issues that make both, `monitorenter` and `monitorexit` not concurrency invariant: First, if either `monitorenter` or `monitorexit` is moved into the concurrent region, as shown in Figure 4.9(b), the `monitorenter` and `monitorexit` operations are potentially executed by different threads. Consequently, the Java VM throws an `IllegalMonitorStateException`, as described in the JLS.

Second, even if both, `monitorenter` as well as `monitorexit` are moved into the same concurrent region, as illustrated in Figure 4.9(c), the happens-before relationship between the `monitorenter` operation and the `cStart` node is removed. Removing this happens-before relationship changes the inter-thread semantics of the parent thread. For example, the code in Figure 4.9(c) unconditionally starts the concurrent region, while the code in Figure 4.9(a) must first acquire the lock to start the concurrent region.

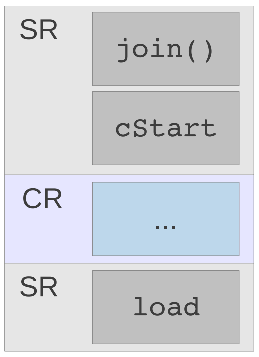
wait(), notify(), and notifyAll()

`wait()`, `notify()`, and `notifyAll()` are low-level communication primitives between threads. `wait()` is not a concurrency invariant operation, since `wait()` is implemented as a method call that cannot be inlined. Note that a method call changes the control flow. Furthermore, `wait()` requires that the thread that calls `wait()` owns the monitor of the object on which the thread calls the `wait()` function. If the JIT compiler moves the `wait()` call into the 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.

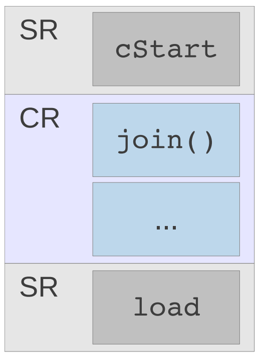
`notify()` and `notifyAll()` are not concurrency invariant for similar reasons as `wait()`. Both calls are implemented as a method calls. Furthermore, both calls require that the thread which performs the call owns the corresponding monitor.

Starting and joining threads

Starting a new thread is not concurrency invariant, since Java implements the action that starts a thread as a method call. As discussed earlier, a method call is per definition not concurrency invariant. Furthermore, starting a new thread involves memory allocation, which can lead to an uncaught exception in the child thread. The join operation blocks until the thread on which the join operation is called terminates. Java implements the join operation as a method call. Consequently, the join operation is not concurrency invariant. Even if join would not be implemented as a method call, moving join into a concurrent region changes the inter-thread semantics of the parent thread and the child thread, as shown in Figure 4.10.



(a) Original program.



(b) Reordering join().

Figure 4.10: Effects of reordering the join operation.

Figure 4.10(a) shows the original program and Figure 4.10(b) shows a variant of the original program in which the JIT compiler moved join operation into the concurrent region. It is easy to see that moving the join operation changes the inter-thread semantics of the parent thread. In Figure 4.10(b) the join operation is executed unconditionally by the child whereas in the original program the parent threads blocks until the join operation finishes successfully. If the load operation that is located in the sequential region that follows the concurrent region in program order accesses a field that is written by the thread on which the parent thread performs the join operation in the original version, moving the join operation to the child thread introduces a data race that does not exist in Figure 4.10(a). Moving the join operation removes the happens-before relationship between the join operation and the subsequent load operation.

4.4.3 Examples of concurrency invariant reorderings

This section provides several examples of concurrency invariant intra-thread actions and concurrency-invariant inter-thread actions.

Concurrency invariant intra-thread actions

Figure 4.11 shows three examples each of which contains concurrency invariant intra-thread actions. The operations marked with (*) are concurrency invariant to a particular concurrent region.

Initially, x == y == 0; x and y are normal fields

Example 1	Example 2	Example 3
0: `r0 = x;`	0: `r0 = x;`	0: `r0 = x;`
1: `r1 = y;`	1: `r1 = r0 + 1;`	1:*`r1 = r0 + 1;`
2:*`r2 = r0 + r1;`	2: `cStart;cBarrier;`	2: `cStart; cBarrier;`
3:*`r3 = r2 + 1;`	3: `x = r1 + 2;`	3: `x = r1 + 1;`
4: `cStart;cBarrier;`	4: `cBarrier;cEnd;`	4: `cBarrier; cEnd;`
5: `r4 = r2 + r3;`	5:*`r2 = r1 + 1;`	5: `cStart; cBarrier;`
6: `cBarrier;cEnd;`	6: `cStart;cBarrier;`	6: `y = r0 + 2;`
7: `return;`	7: `y = r2 + 1;`	7: `cBarrier; cEnd;`
	8: `cBarrier;cEnd;`	8: `return;`
	9: `return;`	

Figure 4.11: Examples of concurrency invariant intra-thread actions.

Operation 0 and operation 1 in Example 1 are inter-thread actions, since the `x` and `y` are normal field loads. The calculation of $(r3 = r2 + 1) \xrightarrow{C_{inv}} \{\mathcal{C}\}$, since the addition of two local variables cannot throw an exception and there is no use of $r3$ after the concurrent region. Similarly, $(r2 = r0 + r1) \xrightarrow{C_{inv}} \{\mathcal{C}\}$, because the addition cannot throw an exception and there is no use of $r2$ after the concurrent region.

Example 2 contains two concurrent regions: $\mathcal{C}_1$ ranges from line 2 - line 4 and $\mathcal{C}_2$ ranges from line 5 - line 8. Example 2 contains one concurrency invariant instruction: $(r2 = r1 + 1) \xrightarrow{C_{inv}} \{\mathcal{C}_2\}$. The reason is that $r2$ is only used in $\mathcal{C}_2$. Operation 1 in example 2 is not concurrency invariant to $\mathcal{C}_1$, since there exists a use of $r1$ after $\mathcal{C}_1$ (operation 5).

Example 3 contains two concurrent regions: $\mathcal{C}_1$ ranges from line 2 - line 4 and $\mathcal{C}_2$ ranges from line 5 - line 7. Example 3 contains read and write operations performed by different threads. The read of field `x` in line 0 and the write to field `x` in line 3 are correctly synchronized, because the accesses are ordered by a happens-before relationship. Example 3 contains one concurrency invariant instruction: $(r1 = r0 + 1) \xrightarrow{C_{inv}} \{\mathcal{C}_1\}$, since there is no use of r1 after $\mathcal{C}_1$.

Concurrency invariant inter-thread actions

Figure 4.12 contains three example of concurrency invariant inter-thread actions. Concurrency-invariant actions are marked with (*).

Initially, x == y == 0; x and y are normal fields

Example 4	Example 5	Example 6
0:*`r1 = x + 1;`	0:*`r1 = x + 1;`	0:*`x = 1;`
1: `r2 = y + 1;`	1:*`r2 = y + 1;`	1: `y = 2;`
2: `cStart;cBarrier;`	2: `cStart;cBarrier;`	2: `cStart;cBarrier;`
3: `call f(r1+r2)`	3: `x = r1 + 2;`	3: `call f(x+1);`
4: `cBarrier;cEnd;`	4: `cBarrier;cEnd;`	4: `cBarrier;cEnd;`
5: `y += 1;`	5:*`y++;`	5: `cStart;cBarrier;`
6: `cSync;`	6: `cStart;cBarrier;`	6: `y++;`
7: `return;`	7: `y = r2 + 1;`	7: `cBarrier;cEnd;`
	8: `cBarrier;cEnd;`	8: `cSync;`
	9: `cSync;`	9: `return;`

Figure 4.12: Examples of concurrency invariant inter-thread actions.

Example 4 contains one concurrent region: $\mathcal{C}$ ranges from line 2 - line 4. Example 4 contains one concurrency invariant instruction. $(r1 = x + 1) \xrightarrow{C_{inv}} \{\mathcal{C}\}$. Line 0 and line 1 can be reordered, since reordering two independent load operations is legal in the JMM. $(r1 = x + 1)$ can be moved into $\mathcal{C}$, since `x` is guaranteed to be accessed only in $\mathcal{C}$ until the next matching `cSync` CIR-node, which is defined in line 6. Note that reordering operation 0 with operation 1 is an intra-region reordering. $(r2 = y + 1)$ is *not* concurrency invariant, because there is a write access to `y` in line 5. Moving $(r2 = y + 1)$ introduces a data race that is not present in the original version. In the original version, the read and the write access of field `y` are ordered by program order. If the JIT compiler moves $(r2 = y + 1)$ into the concurrent region, the JIT compiler introduces a data race, because there is no ordering guarantee between the child thread and the parent thread. Note that moving $(r1 = x + 1)$ into the concurrent region reduces the number of parameters that are passed from the sequential region to the concurrent region from two ($r1$ and $r2$) to one ($r2$).

Example 5 contains two concurrent regions: $\mathcal{C}_1$ ranges from line 2 - line 4 and $\mathcal{C}_2$ ranges from line 6 - line 8. Example 5 contains three concurrency invariant instructions. $(r1 = x + 1) \xrightarrow{C_{inv}} \{\mathcal{C}_1\}$, $(r2 = y + 1) \xrightarrow{C_{inv}} \{\mathcal{C}_2\}$, and $(y{+}{+}) \xrightarrow{C_{inv}} \{\mathcal{C}_2\}$. Similar to Example 4, operations 0 and 1 can be reordered, and $(r1 = x + 1)$ can be moved into $\mathcal{C}_1$. $(y{+}{+}) \xrightarrow{C_{inv}} \{\mathcal{C}_2\}$, because the only access to y is in $\mathcal{C}_2$ until the next matching `cSync` CIR-node. Finally, $(r2 = y + 1) \xrightarrow{C_{inv}} \{\mathcal{C}_2\}$, because $(r2 = y + 1)$ can be moved after $\mathcal{C}_1$. As a result, there is no access to y before the next matching `cSync` CIR-node.

Example 6 contains two concurrent regions. $\mathcal{C}_1$ ranges from line 2 - line 4 and $\mathcal{C}_2$ ranges from in line 5 - line 7. Example 6 contains one concurrency invariant instruction: $(x = 1) \xrightarrow{C_{inv}} \{\mathcal{C}_1\}$, because there is no access to x until the next matching `cSync` CIR-node. Without performing inter-procedural analysis, the JIT compiler cannot determine whether y is accessed in the call to $f()$. If y is read or written to in $f()$ moving y to either $\mathcal{C}_1$ or $\mathcal{C}_2$ introduces a data race that does not exist in the original version.

4.5 Summary

This chapter presented the design of concurrent statements, concurrent methods, and synchronization statements. These three simple source language extension allow a programmer to define and synchronize parallelism at the statement level. Furthermore, this chapter introduced concurrent regions, an explicit representation of concurrent statements/methods in the JIT compiler of a Java VM. Concurrent statements/methods, as well as concurrent regions extend the concurrency interface between the source language and the Java VM so that the high-level information provided by the concurrent statement is made accessible to the Java VM.

Based on the high-level information, this chapter introduced concurrency invariant intra-thread actions and concurrency invariant inter-thread actions. Concurrency invariant actions allow the JIT compiler to change the parallel code granularity by moving instructions into a concurrent region. Moving instructions into a concurrent region increases the possibility to merge concurrent region, which is discussed in the next chapter.

5 System architecture and implementation

This chapter describes the implementation of concurrent statements, concurrent methods, and synchronization statements in the context of the Java platform. Concurrent statements and synchronization statements require modifications to the static compiler, i.e., the compiler that translates Java source code to Java bytecode. The reason is that the current Java Language Specification (JLS) [40] does not support the annotation of statements.

Section 5.1 describes the static compilation of concurrent statements and synchronization statements. Section 5.2 describes the implementation of the new source language constructs in the Jikes Research Virtual Machine (Jikes RVM) [11]. Like any modern Java Virtual Machine (Java VM), the Jikes RVM is equipped with an Adaptive Optimization System (AOS) that dynamically optimizes the running application by recompiling frequently executed methods. Section 5.3 describes extensions to the AOS of the Jikes RVM that enable the Jikes RVM to dynamically optimize the execution of concurrent statements and concurrent methods. In particular, the AOS can switch between the sequential and parallel execution of concurrent statements and concurrent methods. Furthermore, the AOS can merge concurrent statements and concurrent methods. Merging changes the task granularity of the application as provided in the source code. Section 5.4 presents the integration of a thread pool into the Java VM. Finally, Section 5.5 discusses implementation alternatives, i.e., the implementation of concurrent statements/methods and synchronization statements in a runtime system other than the Java VM.

5.1 Static compilation

5.1.1 Static compilation of concurrent statements

The current JLS does not support annotations at the statement level. Annotations can only be put at field declarations and method declarations. There exists a Java

Specification Request (JSR)[1] that proposes to extend Java annotations such that types and statements can be annotated. However, until annotating statements is a standard feature of the Java language, a simple preprocessor that compiles concurrent statements and synchronization statements to standard Java code can be used to make the Java VM aware of statement annotations.

The preprocessor can translate concurrent statements to Java code that is compatible with the JLS is *outlining*. Outlining copies the annotated statement to a new method (a method that is generated by the preprocessor) and annotates the method with `@Concurrent`. Instead of executing the annotated statement directly, a call to the new method, which contains the body of the statement, replaces the concurrent statement. Outlining is a well-established technique to implement parallelism. For example, the GNU Compiler Collection (GCC) uses outlining to implement the OpenMP [4] standard. We use the same technique due to the simple implementation.

A potential drawback of outlining is that the call to the generated method incurs a running-time overhead compared to having the statement inlined. Furthermore, outlining limits the optimization opportunities of the just-in-time compiler (JIT compiler), because JIT compilers perform many optimizations at the method-level granularity. Since an annotated statement is moved to a separate method, the context in which the annotated statement is defined is not visible to the JIT compiler. However, our implementation enables the JIT compiler to eliminate the running-time overhead by inlining the outlined statement. Furthermore, inlining reestablishes the optimization opportunities for the JIT compiler.

Figure 5.1 illustrates how the preprocessor translates a concurrent `for` statement to standard Java code.

```
class X {
  void foo() {
    int r0 = x, r1 = 0;
    @Concurrent
    for (; r1 < 10; r1++) {
      print(r0+r1);
    }
  }


}
```

(a) Concurrent `for` statement.

```
class X {
  void foo() {
    int r0 = x; r1 = 0;
    for (; r1 < 0; r1++)
      __c_foo_0(r0,r1);
  }
  @Concurrent
  final private
  void __c_foo_0(int r0, int r1) {
    print(r0+r1);
  }
}
```

(b) Outlined concurrent `for` statement.

Figure 5.1: Static compilation of concurrent statements.

Figure 5.1(a) shows function `foo()`, which contains a concurrent `for` statement. Recall that a concurrent `for` statement allows the Java VM to execute all loop iterations in parallel. The concurrent `for` statement in Figure 5.1(a) can be processed by

[1]http://code.google.com/p/jsr308-langtools/wiki/AnnotationsOnStatements

up to ten separate threads. Figure 5.1(b) shows to code of method `foo()` to which the preprocessor translates the code in Figure 5.1(a). The preprocessor replaces the body of the concurrent `for` statement by a call to `__c_foo_0(int x, int y)`. The preprocessor annotates the generated method with `@Concurrent` and copies the body of the `for` loop to the body of the generated method.

The generated method has two integer parameters `r0` and `r1`. The preprocessor performs a simple liveness analysis to determine all local variables that are used in the body of the concurrent statement before they are defined. The values of these variables are passed as parameters to the generated method. Figure 5.1(b) illustrates clearly why changes to local variables that are performed inside the body of a concurrent statement are not visible outside the concurrent statement. The preprocessor defines the generated method as `private`. As a result, a subclass of `X` cannot accidentally override the generated method.

5.1.2 Static compilation of synchronization statements

The preprocessor translates synchronization statements to empty method calls that are annotated with the `@Sync` annotation. The synchronization statement body follows the generated method call in program order.

5.2 Extensions to the JIT compiler

This section describes the dynamic compilation system of the Jikes RVM and describes how the JIT compiler translates concurrent methods and synchronization methods to machine code. Note that the preprocessor translates all statement annotations to function calls that are annotated with either `@Concurrent` or `@Sync`. As a result, statement annotations are not directly contained in the bytecode.

5.2.1 The Jikes RVM compilation system

The dynamic compilation system of the Jikes RVM consists of two separate compilers: a baseline compiler and an optimizing compiler. The baseline compiler is a fast compiler that translates bytecode to machine code without performing time-consuming analyses and optimizations. Unlike other Java VM implementations, such as the Hotspot Java VM [51], the Jikes RVM has no bytecode interpreter. The AOS [14] of the Jikes RVM identifies hot methods (methods in which the application spends a significant amount of time) and triggers a recompilation of the hot method with the optimizing compiler. The optimizing compiler has three optimization levels among which the AOS can choose. In general, higher optimization levels produce faster code but consume more compilation time than lower optimization levels.

The optimizing compiler uses three different intermediate representations (IRs) to compile bytecode to machine code. Initially, the optimizing compiler translates Java bytecode to the High-level IR (HIR). The bytecode to HIR transformation pass replaces the Java stack by virtual registers. HIR is independent of the target architecture. Consequently, the optimizing compiler applies architecture independent code transformations like loop unrolling, common subexpression elimination, or redundant load elimination at the HIR. The parallel code optimizations that are described in Chapter 6 are also performed at HIR, since the parallel code optimizations do not need architecture-specific details that are available in the two lower-level IRs.

The optimizing compiler translates the HIR to the low-level IR (LIR). LIR adds details about the object layout to the IR. Furthermore, LIR contains architecture-specific information that enables the optimizing compiler to perform optimizations like instruction scheduling. The optimizing compiler translates the LIR to the machine-level IR (MIR). MIR is similar to the machine code of the target architecture.

5.2.2 Dynamic compilation of concurrent statements

As described in Section 5.1, the preprocessor generates a synthetic method for each concurrent statement and outlines the body of the concurrent statement to that method. Consequently, the JIT compiler is only exposed to annotated method declarations. The optimizing JIT compiler first compiles bytecode to the Concurrency-aware Intermediate Representation (CIR), which extends HIR by four IR-nodes: `cStart`, `cEnd`, `cBarrier`, and `cSync`.

The rules for adding the four CIR nodes are as follows: The bytecode to CIR translation pass checks every method call bytecode whether the target is a concurrent method. This information is contained in the Java binary (`.class` file). If the JIT compiler discovers a concurrent method, the JIT compiler emits (i) a `cStart` CIR-node, (ii) a `cBarrier` CIR-node, (iii) the method call IR node, (iv) a `cBarrier` CIR-node, and (v) the `cEnd` CIR-node. Instead of emitting the method call IR node, the JIT compiler can inline the body of the concurrent call. The method body of the inlined function is then contained between the two `cBarrier` CIR-nodes. Methods that are annotated with `@Sync` are translated to CIR by inserting a `cSync` CIR-node before the method call.

All compiler optimizations that are designed for HIR can be directly applied to CIR without having to modify the individual optimization passes. Examples for HIR-based compiler optimizations are: Static Single Assignment (SSA)-based compiler optimizations like load elimination [32] and global code placement. Non-reordering optimizations like copy/constant propagation or common subexpression elimination can be safely applied across concurrent regions and hence across thread boundaries. Even instruction reordering optimization passes need no modifications to ensure that non-concurrency invariant instructions are not moved out of a concurrent region or into a concurrent region. `cStart`, `cEnd`, `cBarrier`, and `cSync` are given the semantics of non-inlinable function calls; the JIT compiler does not reorder instructions across these CIR-nodes. The JIT compiler uses the CIR to perform concurrent region merging and expends CIR to HIR before the translation to LIR.

The CIR expansion pass converts the four CIR nodes to HIR nodes. In particular, CIR expansion needs to handle inlined and non-inlined concurrent calls.

CIR expansion of non-inlined concurrent calls

Consider Figure 5.2, which illustrates the translation from CIR to HIR.

```
1  foo:
2    cStart;
3    cBarrier;
4    call __c_foo_0(r0,r1);
5    cBarrier;
6    cEnd;
```

(a) CIR.

```
1  foo:
2    o = __new_task_object();
3    o.i[0] = r0;
4    o.i[1] = r1;
5    o.stub_code = __get_stub(__c_foo_0);
6    call __put_thread_pool(o);
```

(b) Expanded CIR (HIR).

```
1  run:
2    call this.stub_code;
3    __clear_scratch();
```

(c) Task object.

```
1  __stub_c_foo_0:
2    o = __get_task_object();
3    r0 = o.i[0];
4    r1 = o.i[1];
5    o.ret[0] = call __c_foo_0(r0,r1);
```

(d) Stub method body.

Figure 5.2: Dynamic compilation of concurrent regions.

Figure 5.2(a) shows the CIR of method `foo()`, which contains one concurrent region that contains one method call that takes two parameters. Figure 5.2(b) shows the HIR of `foo()`. The JIT compiler inserts a series of instructions: First, the JIT compiler inserts the call to the `__new_task_object()` function, which is a Java VM-internal function that returns a new task object. Every instance of a concurrent region requires the construction of a task object. Every task object has the same type and implements the `Runnable` interface. Consequently, a standard Java thread pool implementation can execute the task object.

The inserted instructions in line 3 and line 4 of Figure 5.2(b) save the values of the parameters of the `__c_foo_0(r0,r1)` to a scratch area, which is local to each task object. The scratch area has an array for each primitive type. In the above example, `r1` and `r2` are of type `int`. Consequently, both parameters are stored to the `int` array of the scratch area. Object references are stored to an array of type `Object` in the argument save area. Storing object references as a sequence of bytes does not work for Java VMs that use a copying garbage collector, since the garbage collector must be aware of object references. Otherwise, the stored reference is invalid after the garbage collector moves an object to a different location in memory. The call to the `__get_stub()` function in line 5 returns the address of the first instruction of the stub method, which sets up the call to `__c_foo_0(r0,r1)`. The JIT compiler generates a separate stub method for every concurrent method declaration. However, different call sites of a concurrent method use the same stub method. Finally, the call in line 6 puts the task object to the Java VM-internal thread pool for execution.

Figure 5.2(c) shows the `run()` method of the task object. Line 2 issues a call to the `__stub_c_foo_0()` method, which is shown in Figure 5.2(d). The call in line 3 clears the scratch area, which stores object references that need to be passed to the concurrent region. Clearing the object references avoids potential memory leaks. Note that the child thread executes the `run()` method in Figure 5.2(c). Figure 5.2(d) shows the body of the stub method. The `__get_task_object()` function in line 2 loads the reference to the task object that was generated in Figure 5.2(b). The load instructions in line 3 and line 4 load the parameter values from the scratch area of the task object. Line 5 issues the call to the concurrent method and saves the return value to the scratch area of the task object. Note that the return value save area is also provided as an array. The reason is that a concurrent region can contain more than one concurrent call with a return value. If the concurrent call has no return value, storing the return value is omitted. Function `foo()` is executed by the parent thread while the `run()` method of the task object as well as the stub method can be executed by the parent thread but also by the child thread.

CIR expansion of merged concurrent regions

The expansion of a merged concurrent region, i.e., a concurrent region that contains more than one method call works analogous to the expansion as described above. For example, assume that function `foo()` contains two calls to `__c_foo_0()`. Figure 5.3(a) illustrates the expanded CIR of `foo()` and Figure 5.3(b) illustrates the generated stub method body. The `run()` method of the task object does not change.

```
foo:
  o = __new_task_object();
  o.i[0] = r0;
  o.i[1] = r1;
  o.i[2] = r2;
  o.i[3] = r3;
  o.stub_code =
    __get_stub(__c_foo_0_x_2);
  call __put_thread_pool(o);
```

(a) Expanded CIR.

```
__stub_c_foo_0_x_2:
  o = __get_task_object();
  r0 = o.i[0];
  r1 = o.i[1];
  r2 = o.i[2];
  r3 = o.i[3];
  o.ret[0] = call __c_foo_0(r0,r1);
  o.ret[1] = call __c_foo_0(r2,r3);
```

(b) Stub method body.

Figure 5.3: Dynamic compilation of merged concurrent regions.

As shown in Figure 5.3(a), there is only one call to the `__new_task_object()` function and the `__put_thread_pool()` function. Both function calls that are contained in the concurrent region require the construction of only one task object. As result, the parallel overhead of having two calls in one concurrent region is smaller compared to having each call contained into a separate concurrent region. The values of the arguments to both function calls are copied to the scratch area of the task object in line 3 - line 6. The `__get_stub()` function returns a different address than in Figure 5.2, since the stub function must initialize two method calls. Note that the `_x_2` suffix of the argument of the `__get_stub()` function indicates that `__c_foo_0` is contained twice in the concurrent region.

The stub method body of the merged concurrent region loads the reference to the task object in line 2. The argument values of the method calls are initialized in line 3 - line 6. Finally, line 7 and line 8 contain the two calls to the `__c_foo()` function. The return values are stored in different fields of the argument save area of the task object.

CIR expansion of inlined concurrent calls

The CIR expansion pass converts inlined concurrent calls to HIR by "outlining" all instructions of the concurrent region (see Section 4.2.2) to a stub method. Local variables that are used but not defined in the concurrent region are identified using liveness analysis [69]. Similar to passing the values of the arguments of a concurrent call to the stub method, the expansion pass passes the values of live variables via the scratch memory of the task object.

Nested concurrent regions

The current implementation does not support nested concurrent regions. If the Java VM detects that a child thread tries to enter a concurrent region, the child thread throws an `NestedConcurrentRegion` exception. However, support for nested concurrent regions can be added to the Java VM without having to change the semantics of the CIR nodes.

Subtype polymorphism

The JIT compiler resolves the target of each concurrent method and checks the the target is annotated with `@Concurrent`. If a normal method is overridden by a concurrent method, the JIT compiler considers the method as a concurrent method. On the other hand, if a concurrent method is overridden by a normal method, the JIT compiler considers the method as a normal method.

5.2.3 Dynamic compilation of synchronization statements

The JIT compiler expands the `cSync` CIR-node to HIR by inserting the call to the Java VM-internal method that implements the synchronization of concurrent regions. To synchronize concurrent regions, each parent thread maintains a list of tasks that the parent thread spawned. The parent thread checks for completion of the issued tasks by iterating through the list and checking the status of each task. The reference to the task is of type `Future`; calling the `get()` method on the task reference blocks until the tasks has finished.

The expansion of synchronization statements needs to implement the return value of a concurrent call. To implement return values for concurrent calls, the JIT compiler inserts code that initializes the return value of the concurrent call just after the call to the Java VM-interval method that implements the synchronization of tasks. Note that there cannot be data races on local variables, since the variable that holds the return value of a concurrent call is `final`.

If a concurrent region terminates abnormally (i.e., throws an unchecked exception) the Java VM removes the parallel tasks from the list of spawned tasks that is maintained by the parent thread. As a result, the `sync` statement does not incur a deadlock. Furthermore, the Java VM increments the counter value that is returned by the `Thread.getError()` function by one.

5.3 Concurrent region profiling

This section describes the architecture of the AOS that is part of the Jikes RVM in Section 5.3.1. Section 5.3.2 presents extensions to the existing AOS that enable concurrent region profiling. Finally, Section 5.3.2 describes the implementation of dynamic concurrent region merging.

5.3.1 The Jikes RVM AOS

The Jikes RVM uses a sampling-based profiler [97] to identify hot methods. A method is *hot* if the application spends a certain amount of time in the method. The exact threshold that identifies a method as hot is Java VM-specific. To estimate the time that is spent in a method, the Jikes RVM samples the stack of every thread at a regular interval and identifies the current method based on the value of the instruction pointer. If a method is sampled, a hotness counter that is associated with every method is incremented by one. If the hotness counter reaches a certain threshold, the method is submitted to the recompilation controller, which decides - based on the collected profile - if the method is recompiled and at what optimization level. The recompilation is done by a separate thread that can run in parallel with the program execution. Figure 5.4 illustrates the basic architecture of the multi-stage recompilation system.

Initially, the baseline compiler translates bytecode to machine code. The baseline compiler is a fast template-based code generator that incurs only a small compilation time overhead. The sampling profiler profiles the baseline-compiled code and reports hot methods to the recompilation controller. The recompilation controller evaluates

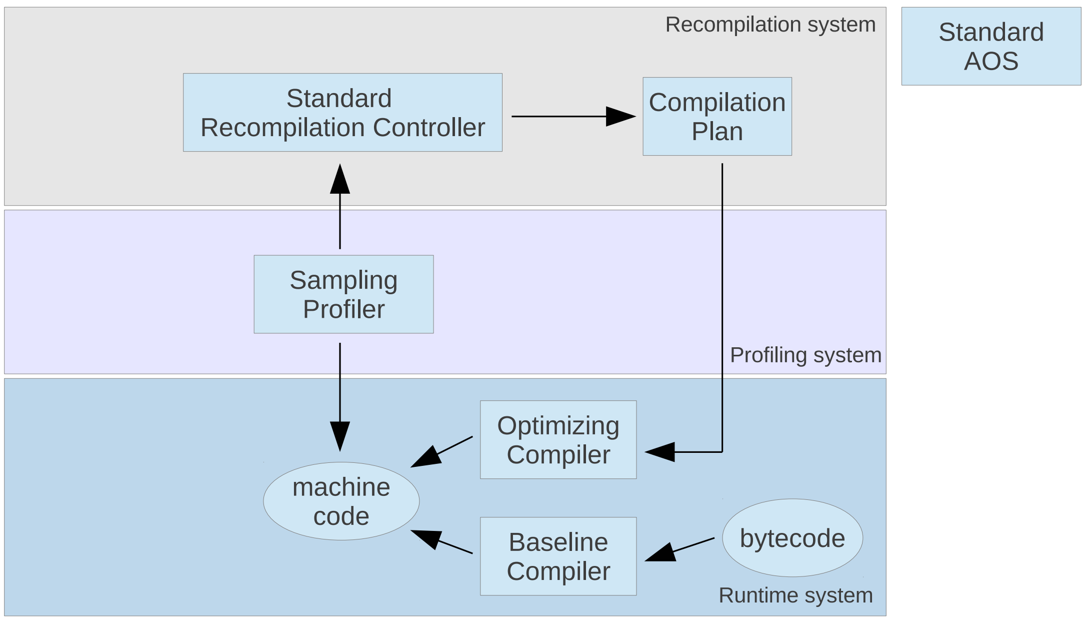


Figure 5.4: Multi-level adaptive optimization system architecture.

the profile and selects a compilation plan, which describes how the method should be recompiled. For example, the compilation plan contains the optimization level at which the method will be recompiled. Furthermore, the compilation plan contains an instrumentation plan, which instructs the JIT compiler to instrument the code to get a more fine-grained profile of the method. For example, the instrumentation plan can instruct the JIT compiler to instrument loop back edges. As a result, the instrumentation plan delivers the exact number of executed loop iterations.

5.3.2 Profiling concurrent regions

There are two important performance characteristics of a concurrent region profile: the *execution time* (T_{exec}) and *number of invocations* (I). 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 in order 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 RVM sampling profiler can only provide an estimate of how much time the application spent in a particular method (T_{total}). Note that T_{total} does not enable the AOS 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 the execution time of a concurrent region as well as the total number of invocations, we add an instrumenting profiler to the profiling system of the Jikes RVM. Figure 5.5 shows the architecture of the concurrency-aware AOS.

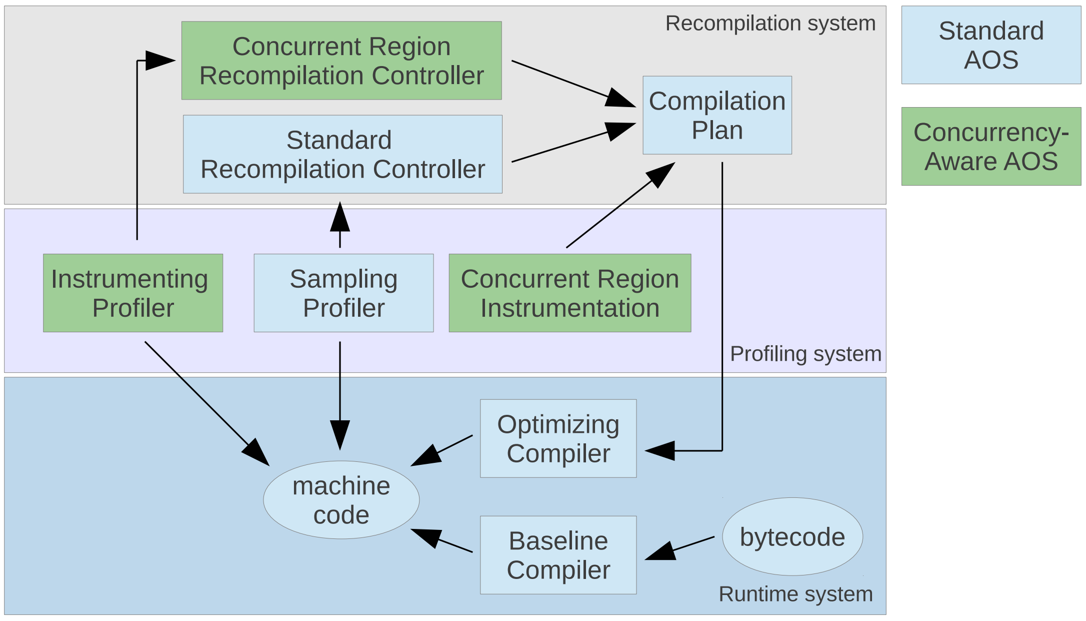


Figure 5.5: Multi-level adaptive optimization system architecture.

Figure 5.5 shows that we extend the profiling system of the concurrency-aware AOS by two components: the *instrumenting profiler* and the *concurrent region instrumentation plan*. Furthermore, we extend the recompilation system by the *recompilation controller*. The subsequent sections describe the three new components in more detail.

Concurrent region instrumentation

Figure 5.6(a) shows a simple instrumentation pattern that enables the exact measurement of the execution time of a concurrent region. Line 1 stores the result of the read time stamp counter (`rdtsc`) instruction in `r1`. Line 2 contains the code of the concurrent region. Line 3 stores the result of the `rdtsc` instruction to `r2`. `r1` and `r2` are 64-bit registers. Line 4 provides the elapsed time (`r2 - r1`) to the instrumenting profiler. Note that a correct usage of the `rdtsc` instruction requires to pin threads to particular cores, since the clocks of different processor sockets are, in general, not synchronized.

As illustrated in Figure 5.6(a), concurrent region instrumentation has two sources of overhead. The first source of overhead is the execution of the `rdtsc` instruction. The overhead of the `rdtsc` instruction is negligibly, since the `rdtsc` instruction is

(i) not serializing and (ii) not ordered with other instructions. I.e., the completion of `rdtsc` does not guarantee that previous instructions have retired. Also, subsequent instructions can begin execution before the `rdtsc` instruction completes. The second source of overhead is related to storing and maintaining the gathered profile data (line 4 in Figure 5.6(a)). As we show in Section 7.2, storing and maintaining the profile data can cause a slowdown of 100X or more compared to an unprofiled execution.

To provide a precise measurement of T_{exec} that incurs a low profiling overhead the instrumenting profiler records samples only at a certain interval. The determination of this *sample record interval* is explained in Section 7.2. Figure 5.6(b) illustrates the sampling instrumentation.

```
1  r1 = rdtsc;
2  //concurrent region
3  r2 = rdtsc;
4  record(r2 - r1);
```

(a) Naive instrumentation.

```
1  r1 = rdtsc;
2  //concurrent region
3  r2 = rdtsc;
4  if (r2 - thread.last_record > THRESHOLD) {
5    record(r2 - r1);
6    thread.last_record = r2;
7  }
```

(b) Sampling instrumentation profiling.

Figure 5.6: Profiling of concurrent regions.

Similar to the naive instrumentation, sampling instrumentation uses the `rdtsc` instruction to measure the execution time of the concurrent region. However, the elapsed time is only recorded at certain intervals. The `if` statement in line 4 checks if the time since the last record (t_{rec}) is larger than `THRESHOLD`. To perform this check, each thread has a field, `last_record`, that holds the value of the time stamp counter when the last sample was recorded. The `last_record` field is initialized to `0` when a thread is created. If the `if` statement in line 4 evaluates to true, the execution time of the concurrent region is recorded and the `last_record` field is set to the value of `r2`. Otherwise, the sample is not recorded.

A concurrent region is not instrumented if the concurrent region neither contains a method call nor a loop. Consequently, the concurrent region is never compiled to parallel code. The execution time of straight line code is not likely to justify the overhead from offloading the code to a separate thread. There is a tradeoff between the overhead of the sampling instrumentation as described above and the precision of the measurement of T_{exec}. If `THRESHOLD` is too large, the number of recorded samples is small. Consequently, if T_{exec} varies heavily upon every invocation, the sampled execution time can be imprecise. However, if `THRESHOLD` is too small, the profiling overhead can be significant.

To measure the number of concurrent region invocations (I), the JIT compiler adds a local variable that is incremented in every concurrent region that is executed in parallel. For merged concurrent regions, the counter is incremented only once. Similar to the `record()` function in Figure 5.6, the JIT compiler adds a function call `record_invocations()` as the last operation before the method returns to the caller. This function provides I to the instrumenting profiler.

Maintaining concurrent region performance samples

Figure 5.7 illustrates the maintenance of the performance samples.

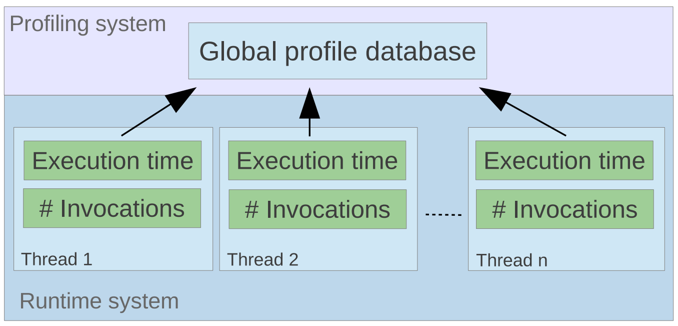


Figure 5.7: Multi-level adaptive optimization system architecture.

Every child thread stores the execution time samples and the number of concurrent region invocations in a data structure that is only accessible to the child thread. The ID of the concurrent region in combination with the ID of the method is used as a key for a hash map that holds the reference to the circular buffer that stores a series of 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. Reporting data to the global database incurs an additional overhead, as discussed later in this section.

In the current implementation, the child thread computes the arithmetic average. The report threshold starts with a value of 1 and increases up to 100 in strides of 10. This reporting strategy ensures that performance data of newly discovered concurrent regions are reported immediately and frequently. At the same time, the execution time of concurrent regions that are profiled for a longer time is reported less frequently. Each thread maintains the number of concurrent region invocations in a similar manner. There is an additional condition for a child thread to report a performance sample to the global database. The computed arithmetic average must differ at least by 30% to the last reported average. Reporting a similar sample provides no new information to the concurrent region recompilation controller. We choose 30% based on empirical evaluation.

The child thread wakes up the concurrent region recompilation controller each time it commits a new sample to the global database. The recompilation controller evaluates the performance characteristic of the entire method that is associated with the new sample. The concurrent region recompilation controller is implemented as a separate thread. As a result, new samples can be evaluated concurrently to the executing application.

Reporting a performance sample to the global sample database incurs an additional overhead, since the reporting thread (the child thread) must acquire a global lock that protects access to the global database. The global lock is necessary since multiple threads can perform updates to the global database at the same time. Furthermore, since the concurrent region recompilation controller is implemented as a separate thread, child threads as well as the concurrent region recompilation controller can read from and write to shared data. The global lock ensures the integrity of the global performance sample database.

5.3.3 Concurrent region merging

Concurrent region merging is done at run-time, since there is a large number of possible executions of merged and non-merged concurrent regions, concurrent regions that are compiled to sequential or parallel code that potentially yields the best performance. The total number of possible assignments of concurrent regions to threads including all possible reorderings is given by the following formula:

$$E = n! \times 2^{n-1}$$

E is the number of possible executions. n is the number of concurrent regions that can be merged. The intuition behind the formula is that mergable concurrent regions can be arranged in an arbitrary order. Re-arranging concurrent regions yields a possible combination of concurrent regions of $n!$. In addition, every permutation has 2^{n-1} merging combinations. The state explosion makes a static compilation of all possible combinations of reorderings and concurrent region merging combinations impractical. For example, $n = 4$ there are $E = 192$ possible combinations.

Figure 5.8 shows the potential merging combinations of one permutation.

(1) $\mathcal{C}_1 \quad \mathcal{C}_2 \quad \mathcal{C}_3 \quad \mathcal{C}_4$

(2) $\boxed{\mathcal{C}_1 \quad \mathcal{C}_2} \quad \mathcal{C}_3 \quad \mathcal{C}_4$

(3) $\mathcal{C}_1 \quad \boxed{\mathcal{C}_2 \quad \mathcal{C}_3} \quad \mathcal{C}_4$

(4) $\mathcal{C}_1 \quad \mathcal{C}_2 \quad \boxed{\mathcal{C}_3 \quad \mathcal{C}_4}$

(5) $\boxed{\mathcal{C}_1 \quad \mathcal{C}_2} \quad \boxed{\mathcal{C}_3 \quad \mathcal{C}_4}$

(6) $\boxed{\mathcal{C}_1 \quad \mathcal{C}_2 \quad \mathcal{C}_3} \quad \mathcal{C}_4$

(7) $\mathcal{C}_1 \quad \boxed{\mathcal{C}_2 \quad \mathcal{C}_3 \quad \mathcal{C}_4}$

(8) $\boxed{\mathcal{C}_1 \quad \mathcal{C}_2 \quad \mathcal{C}_3 \quad \mathcal{C}_4}$

Figure 5.8: Potential merging combinations.

Figure 5.8 shows that the four concurrent regions $\mathcal{C}_1$ - $\mathcal{C}_4$ yield eight merging configurations. Concurrent regions that are contained in the same box are merged.

Representing merged concurrent regions in the JIT compiler

This subsequent definition describes how the JIT compiler represents merged concurrent region(s).

Definition 5 ***(Concurrent region merging)*** *Assume that there are two concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ that reside in the same method. Furthermore, assume that $\mathcal{C}_1$ comes before $\mathcal{C}_2$ in program order. A concurrent region $\mathcal{C}_{12}$ that is composed of $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined as follows:*

- $\mathcal{C}_{12\mathcal{S}} = \mathcal{C}_{1\mathcal{S}} = \mathcal{C}_{2\mathcal{S}}$*: The signature of the method that contains the merged concurrent region is the same as the signature of $\mathcal{C}_1$ and $\mathcal{C}_2$.*
- $\mathcal{C}_{12\mathcal{B}} = \mathcal{C}_{1\mathcal{B}} \oplus \mathcal{C}_{2\mathcal{B}}$*: ($\oplus$ is the append operation). The list of pointers into the bytecode array consists of the pointers of $\mathcal{C}_1$ and $\mathcal{C}_2$.*
- $\mathcal{C}_{12\mathcal{I}} =$ *new unique id.*
- $\mathcal{C}_{12\mathcal{P}} = \mathcal{C}_{1\mathcal{P}} \oplus \mathcal{C}_{2\mathcal{P}}$*.*
- $\mathcal{C}_{12\mathcal{M}} = \mathcal{C}_{1\mathcal{M}} \cup \mathcal{C}_{2\mathcal{M}}$*: $\cup$ is the union operation.*

If there are n concurrent regions that can be merged, the merging can happen iteratively, i.e., the JIT compiler first merges $\mathcal{C}_1$ and $\mathcal{C}_2$, which gives the merged concurrent region $\mathcal{C}_{12}$. The JIT compiler then merges $\mathcal{C}_{12}$ and $\mathcal{C}_3$, which gives $\mathcal{C}_{123}$, and so forth. Definition 6 and Definition 7 in Section 6.1.3 specify the conditions under which concurrent regions can be merged.

5.3.4 Adaptive parallel code generation example

Figure 5.9 illustrates the steps that are involved in compiling a concurrent region to parallel code.

In our execution model, a parallel execution of a concurrent region is considered as an optimization that is applied by the Java VM. Every concurrent region starts executing sequentially ($\textcircled{1}$). If the sampling profiler identifies a hot method that contains a concurrent region and the standard recompilation controller decides to recompile the method, the JIT compiler instruments the concurrent region and compiles the method to sequential code ($\textcircled{2}$). From the instrumented sequential version, the JIT compiler can compile a concurrent region to either parallel code $\textcircled{3a}$ or merge concurrent regions ($\textcircled{3b}$). A concurrent region that is compiled to parallel code can either be recompiled such that concurrent regions are demerged ($\textcircled{4a}$) or merged ($\textcircled{4b}$). Either of the parallel code versions in Figure 5.9 can be compiled to sequential code ($\textcircled{5a}$, $\textcircled{5b}$).

5.4 JVM service extensions

We extend the Java VM with a thread pool. The thread pool provides the Java VM with the opportunity to efficiently execute concurrent regions. Note that the thread

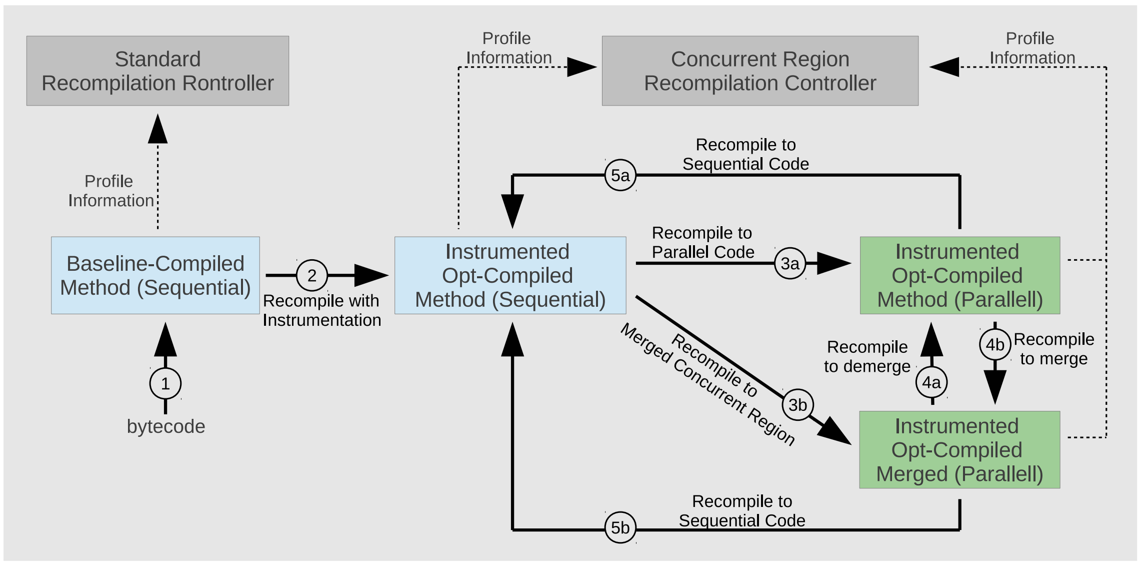

Standard Recompilation Rontroller
Profile Information
Concurrent Region Recompilation Controller
Profile Information
Recompile to Sequential Code
5a
Profile Information
Baseline-Compiled Method (Sequential)
2
Recompile with Instrumentation
Instrumented Opt-Compiled Method (Sequential)
Recompile to Parallel Code
3a
Instrumented Opt-Compiled Method (Parallell)
Recompile to Merged Concurrent Region
3b
Recompile to demerge
4a
4b
Recompile to merge
Instrumented Opt-Compiled Merged (Parallell)
1
bytecode
Recompile to Sequential Code
5b

Figure 5.9: Adaptive parallel code generation.

pool can implement arbitrary scheduling strategies. For example, a specific thread pool implementation can use work stealing [58] to keep the workload balanced. The current Java VM thread pool is an instance of `java.util.concurrent` package with a fixed number of threads. The Java VM-internal thread pool is started at the Java VM startup and uses, by default, n threads, where n is the number of physical cores. However, the user of the Java VM can specify the number of threads by using the `-X:availableProcessors=n` option. Details about the thread pool implementation can be found in the Java documentation.

5.5 Implementation alternatives

This section describes alternative runtime systems that are suitable to implement concurrent statements/methods as well as the parallel code optimizations that are discussed in the next chapter. The two main components that are required for runtime system-centric concurrency management are (i) a JIT compiler and (ii) a profiling system. The Low Level Virtual Machine (LLVM) [56] and the `.NET` [2] platform and provide the basic infrastructure to perform parallel code optimizations at run-time.

5.5.1 LLVM-based implementation

LLVM is a compiler framework that provides dynamic program analysis and transformation for arbitrary programs. LLVM provides high-level program information to compiler transformations at compile-time, link-time, and run-time. LLVM internally uses a low-level code representation in SSA form that is generated by a static compiler. LLVM can be configured to enable dynamic recompilation of hot code regions. Similar to a Java VM, the LLVM compiler framework instruments loops to identify hot paths within the loop (hot trace). Such a hot trace is provided to a JIT compiler that recompiles the hot trace using sophisticated compiler optimizations.

The input to the LLVM compiler framework is the LLVM-IR, which is a low-level RISC-like virtual instruction set IR that is strongly typed using a simple type system. The LLVM-IR consists of only 31 opcodes. Similar to the Java Runtime Environment (JRE) the LLVM concurrency interfaces cannot represent the concurrency structure of the application, i.e., none of the 31 opcodes (or a combination thereof) is able to represent the concurrency structure of parallel code. Like many other compiler frameworks (see Chapter 3) LLVM implements parallelism as a library.

One possibility to enable parallel code optimizations in LLVM is to extend to LLVM-IR to a CIR-like IR that is described in this thesis. For example, the type system can be extended to explicitly represent concurrent statements in the LLVM-IR (LLVM-CIR). To make the concurrency structure of the application available to LLVM, the static compiler must be extended to generate LLVM-CIR. An LLVM version that is OpenMP [4] compatible requires no source language extensions, since OpenMP provides a broad spectrum of directives that can be mapped to CIR (see Section 4.2.3).

The current version (3.1) of Clang [1], which is a compiler front end for C, C++, Objective-C, and Objective-C++ that uses LLVM as a back-end does not yet support the OpenMP standard. If Clang supports OpenMP in a future release, LLVM can perform concurrency-aware optimizations based on profile information. OpenMP is a good candidate, since OpenMP has a memory model [44] that exactly defines the legal code transformations.

5.5.2 `.NET`-based implementation

`.NET` is a software platform developed by Microsoft. The design of the `.NET` platform is similar to the design of the JRE: `.NET` consists of a runtime system, which is called Common Language Runtime (CLR), a large collection of libraries, programming interfaces, and services. The CLR uses a profiling system and a JIT compiler to dynamically optimize hot code regions. A static compiler translates the source program (e.g., C#) to an IR which can be executed by the CLR. The IR is called Common Intermediate Language (CIL). Similar to the Java bytecode, the CIL has no special means to represent the concurrency structure of the application. The `.NET` platform implements parallelism as a library.

C# supports two keywords that can be mapped to concurrent regions in the CIL: `async` and `await` (see Section 4.2.3). Unfortunately, the JIT compiler in the CLR cannot implement parallel code optimizations, since the concurrency interface between C# and CIL is unaware of the two keywords. Similar to the approach that is presented in this thesis, an extension of the concurrency interface between C#, CIL, and CLR enables concurrency-aware optimizations. The main advantage of a C#-based implementation is that no source language modifications are required. Furthermore, CLR has a memory model that defines the semantics of parallel C# applications.

5.5.3 Summary

Although there are several implementation alternatives, we consider using the JRE as most suitable for the following reasons: First, there exist several open source implementations of static Java compilers and Java VMs. Second, Java has a sophisticated memory model, the Java Memory Model (JMM), which allows a concise definition of the CIR. Finally, the concurrency interfaces can be extended with small effort.

6
Parallel code optimizations

This chapter presents optimizations for parallel code that the Java Virtual Machine (Java VM) can perform fully automatically. The presented optimizations are based on the Concurrency-aware Intermediate Representation (CIR) and use profile information collected at run-time. The presented optimizations are designed for non-recursive parallel parallel programs. In particular, this chapter presents two kinds of parallel code optimizations.

First, *parallel code execution optimizations* (PCEOs) aim at finding a configuration for a set of concurrent regions such that executing the concurrent regions incurs low overhead and the application provides a good load balance. Section 6.1.3 presents an algorithm that uses profile information to determine (i) if a concurrent region is executed sequentially, (ii) if a concurrent region is executed in parallel, and (iii) the granularity of concurrent regions. Determining these three conditions at compile time is a hard problem, since there are numerous parameters that are unknown at compile time and that have a significant impact on the execution time of parallel code. For example, the execution time of a parallel task can depend on the program input, which is, in general, unknown at compile time. Furthermore, the hardware architecture of the target platform (e.g., number of cores, cache size, and clock frequency) influence the execution time of a parallel task. In addition, if the application runs in a managed runtime environment like the Java VM, Java VM-specific services like the just-in-time compiler (JIT compiler) [57, 81], dynamic memory allocation [98], and garbage collection influence the execution time of parallel code. The impact of such complex runtime services is extremely hard to estimate [17, 65]. As a result, determining the three options described above for a set of concurrent regions is best done at run-time by the Java VM, which has the most up-to-date information and can react to a changing environment.

Second, *parallel overhead reduction optimizations* (POROs) aim at reducing the overhead of executing a concurrent region in parallel. Section 6.2 discusses two code optimizations that are based on CIR. Unlike the PCEOs, parallel overhead reduction optimizations do not require profile information.

6.1 Parallel code execution optimization

Parallel code execution optimizations (PCEOs) aim at providing an efficient execution for concurrent regions. The presented PCEO (called PCEO for the rest of the thesis) is based on CIR and all optimizations are performed fully-automatically by the JIT compiler at run-time based on profile information. A fundamental element of the PCEO is concurrent region merging. Consider the following definition that provides the conditions under which the merging of two concurrent regions is legal. The merging of two concurrent regions is legal if, and only if, the merging cannot introduce any new behaviors, i.e., behaviors that are not possible in an execution where the concurrent regions are not merged.

Merging non-merged concurrent regions

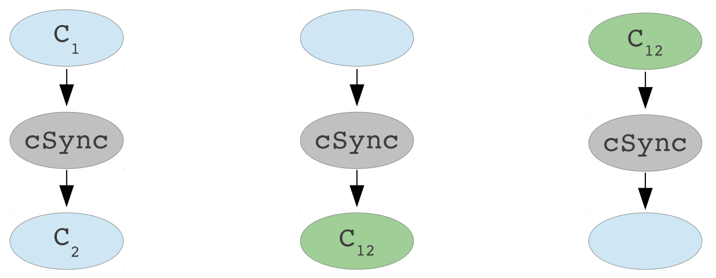
Definition 6 ***(Merging Non-Merged Concurrent Regions)*** *Assume there are two concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ and that $\mathcal{C}_1$ as well as $\mathcal{C}_2$ correspond to a single concurrent statement in the source code. I.e., $\mathcal{C}_1$ and $\mathcal{C}_2$ are not merged with other concurrent regions. Furthermore, assume that $\mathcal{C}_1$ is defined in basic block ($\mathcal{BB}_1$) and $\mathcal{C}_2$ is defined in $\mathcal{BB}_2$. $\mathcal{C}_1$ and $\mathcal{C}_2$ can be merged if and only if:*

1. *$\mathcal{BB}_1 == \mathcal{BB}_2$ and there is no instruction between $\mathcal{C}_1$ and $\mathcal{C}_2$*
2. *all instructions between $\mathcal{C}_1$ and $\mathcal{C}_2$ are concurrency invariant to either $\mathcal{C}_1$ and/or $\mathcal{C}_2$* ***or***
3. *there is no `@Sync` annotated method on any path from $\mathcal{C}_1$ to $\mathcal{C}_2$* ***and***
4. *there is no synchronization action in a sequential region (or concurrent region that is compiled to sequential code) that is on a path from $\mathcal{C}_1$ to $\mathcal{C}_2$* **`and`**
5. *$\mathcal{BB}_1$ dominates $\mathcal{BB}_2$.*

Condition 1: If there is no instruction between $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1$ and $\mathcal{C}_2$ 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 `goto`) between $\mathcal{C}_1$ and $\mathcal{C}_2$. Section 6.1.1 discusses contiguous concurrent region merging in more detail.

Condition 2: If all instructions between $\mathcal{C}_1$ and $\mathcal{C}_2$ are concurrency invariant to either $\mathcal{C}_1$ and/or $\mathcal{C}_2$, the instructions can be moved into $\mathcal{C}_1/\mathcal{C}_2$. As a result, the concurrent regions are contiguous and therefore mergable.

Condition 3: Two concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ cannot be merged if there is a synchronization statement on a path from $\mathcal{C}_1$ to $\mathcal{C}_2$. Figure 6.1 shows an example that illustrates the effects of such a merging. Figure 6.1(a) shows the original program. Figure 6.1(b) shows a variant of the original program in which $\mathcal{C}_1$ and $\mathcal{C}_2$ are merged ($\mathcal{C}_{12}$) and the execution of $\mathcal{C}_{12}$ is issued after the synchronization statement. Such a code transformation introduces a new behavior, since there is no guarantee that the effects of $\mathcal{C}_1$ are visible to other threads before $\mathcal{C}_2$ starts executing. Similarly, the code transformation as shown in Figure 6.1(c) potentially introduces a new behavior, since there is no guarantee that the effects of $\mathcal{C}_1$ are visible to other threads before $\mathcal{C}_2$ starts executing.

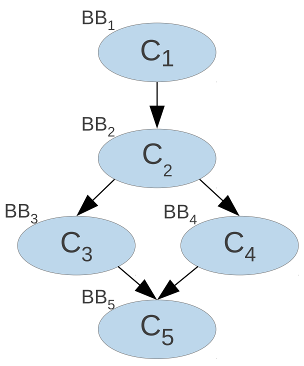


(a) Original program. (b) $\mathcal{C}_1$+$\mathcal{C}_2$. (c) $\mathcal{C}_1$+$\mathcal{C}_2$.

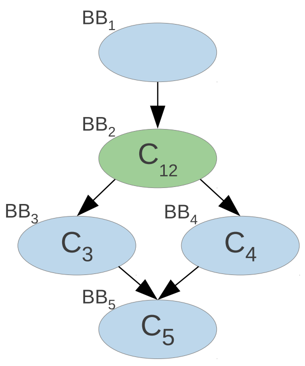
Figure 6.1: Merging of concurrent regions with `cSync`

Condition 4: If there is a synchronization action in a sequential region on a path from $\mathcal{C}_1$ to $\mathcal{C}_2$, merging $\mathcal{C}_1$ with $\mathcal{C}_2$ introduces a new behavior, since the ordering guarantees of the original program are changed. For example, consider the program as given in Figure 6.1. Assume that $\mathcal{C}_1$ consists of a volatile store to a field (`x = 1`) and $\mathcal{C}_2$ consists of the following volatile store (`x = 0`). Furthermore, assume that the `cSync` node is replaced by the following `while` statement: `while (x == 0) `. Initially, `x = 0`. The original program, as illustrated in Figure 6.1(a) guarantees that both concurrent regions as well as any region that follows $\mathcal{C}_2$ in program order are executed. However, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are merged as illustrated in Figure 6.1(b) the merged concurrent region will never be executed, since the volatile store to field `x` is moved after the volatile read in the while loop. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are merged as illustrated in Figure 6.1(c) the `while` statement loops forever. Both behavior are not possible in the original program. As a result, the code transformation is illegal.

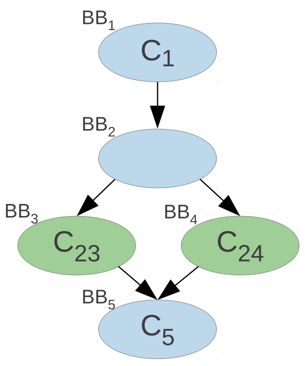
Condition 5: $\mathcal{BB}_1$ must dominate $\mathcal{BB}_2$ so that the merged concurrent region can safely be issued at $\mathcal{BB}_2$. If $\mathcal{BB}_1$ does not dominate $\mathcal{BB}_2$, the concurrent region that is defined in $\mathcal{BB}_2$ is eventually never executed. Consider the examples as shown in Figure 6.2.



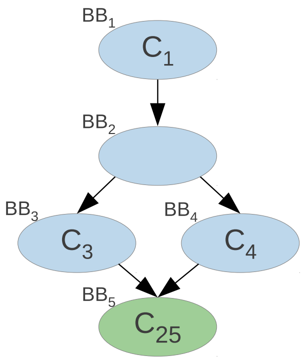
(a) Original program.



(b) $\mathcal{C}_1$+$\mathcal{C}_2$.



(c) $\mathcal{C}_2$+$\mathcal{C}_3$ and $\mathcal{C}_2 + \mathcal{C}_4$.



(d) $\mathcal{C}_2$+$\mathcal{C}_5$.

Figure 6.2: Concurrent region merging examples.

Figure 6.2(a) illustrates the original program. The arrows in Figure 6.2 indicate transitions between basic blocks (BB)s that are triggered in a *sequential region*. I.e., the transition from $\mathcal{BB}_2$ to $\mathcal{BB}_3$ and $\mathcal{BB}_2$ to $\mathcal{BB}_4$ is not determined in $\mathcal{C}_2$. Figure 6.2(b) shows a variant of the original program in which the concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ are merged. $\mathcal{BB}_1$ and $\mathcal{BB}_2$ can be combined to a single basic block. As a result, $\mathcal{C}_1$ and $\mathcal{C}_2$ are contiguous concurrent regions and can therefore be legally merged. Figure 6.2(c) shows an example in which $\mathcal{C}_2$ is merged with $\mathcal{C}_3$ *and* $\mathcal{C}_2$ is merged with $\mathcal{C}_4$. Note that either merging $\mathcal{C}_2$ with $\mathcal{C}_3$ *or* $\mathcal{C}_2$ with $\mathcal{C}_4$ results in an illegal program, since $\mathcal{C}_2$ is then not guaranteed to be executed in any path from $\mathcal{BB}_1$ to $\mathcal{BB}_5$ (violation of condition 5). Finally, Figure 6.2(d) shows a variant of the original program in which $\mathcal{C}_2$ is merged with $\mathcal{C}_5$. This code transformation is legal if $\mathcal{C}_3$ and $\mathcal{C}_4$ are compiled to parallel code. Since $\mathcal{BB}_2$ dominates $\mathcal{BB}_5$, both concurrent regions are guaranteed to be executed. Merging $\mathcal{C}_2$ with $\mathcal{C}_5$ simply determines a particular schedule of execution $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{C}_4$, and $\mathcal{C}_5$.

Merging merged concurrent regions

Definition 7 ***(Merging Merged Concurrent Regions)*** *Assume there are two merged concurrent regions $\mathcal{C}_x$ and $\mathcal{C}_y$. $\mathcal{C}_x$ and $\mathcal{C}_y$ can be merged if and only if every concurrent region that is merged into $\mathcal{C}_x$ is mergable with every concurrent region that is merged into $\mathcal{C}_y$.*

6.1.1 Contiguous concurrent region merging

A simple scenario to merge two concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ is if $\mathcal{C}_1$ and $\mathcal{C}_2$ are *contiguous*. Two concurrent regions are contiguous if there is no instruction between the `cEnd` CIR-node of $\mathcal{C}_1$ and the `cStart` CIR-node of $\mathcal{C}_2$; this implies that $\mathcal{C}_1$ and $\mathcal{C}_2$ are contained in the same basic block. Contiguous concurrent regions can be either defined by the programmer or be generated by previous code transformations. For example, if a concurrent method call is contained in a loop and the JIT compiler unrolls that loop, the JIT compiler generates a set of contiguous concurrent regions. Figure 6.3(a) shows an example with two consecutive concurrent regions.

```
 1  call A
 2  cStart; //cRegion1
 3  cBarrier
 4   call B
 5  cBarrier;
 6  cEnd      //cRegion1
 7
 8  cStart; //cRegion2
 9  cBarrier
10   call C
11  cBarrier;
12  cEnd      //cRegion2
```

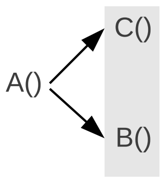
(a) Original program.

```
 1  call A
 2  cStart; //cRegion1
 3  cBarrier
 4   call B
 5
 6
 7  cBarrier
 8
 9
10   call C
11  cBarrier;
12  cEnd      //cRegion1
```

(b) After concurrent region merging.

Figure 6.3: Contiguous concurrent region merging.

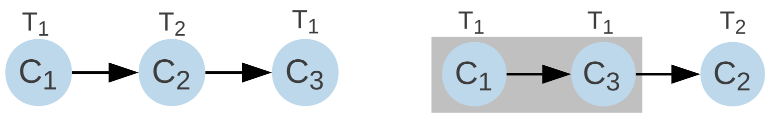
The call to method `A()` is contained in a sequential region and must therefore be executed before $\mathcal{C}_1$. Each of the two concurrent regions (line 2 - line 6 and line 8 - line 12) contains one method call. There is no instruction and no synchronization between $\mathcal{C}_1$ and $\mathcal{C}_2$. Consequently, the following partial order of function calls `A()`, `B()`, and `C()` is legal:



In a legal partial order, function `A()` must have returned before function `B()` and function `C()` can be called. All effects of function `A()` must be visible to both concurrent regions. However, there is no ordering between the calls to the functions `B()` and `C()`. Functions `B()` and `C()` can be executed in parallel. Merging $\mathcal{C}_1$ and $\mathcal{C}_2$ is a legal code transformation, since the merging reduces the number of possible behaviors to one particular partial order and does not introduce any new behaviors. Note that merging $\mathcal{C}_1$ and $\mathcal{C}_2$ is only legal because concurrent statements are serializable. A traditional JIT compiler that correctly implements the Java Memory Model (JMM) must prove that executing the two function calls in a single thread cannot introduce new behaviors. An example of new behavior that can be introduced by merging two concurrent regions that are not serializable is if `B()` waits for function `C()` to complete. In this case the merged program has a deadlock that cannot occur in the original program if function `B()` and function `C()` are executed by different threads.

The JIT compiler can merge the two concurrent regions by removing the `cEnd` CIR-node in line 6 and the `cStart` CIR-node in line 8 of Figure 6.3(a). Figure 6.3(b) shows the code of Figure 6.3(a) after concurrent region merging. Two consecutive `cBarrier` CIR-nodes are redundant and one of the two `cBarrier` CIR-nodes can be optimized. One `cBarrier` CIR node must remain to prohibit the "bait-and-switch" behavior that can be introduced by subsequent compiler passes if the JIT compiler inlines the body function `B()` and function `C()` (see Section 4.3.2).

Contiguous concurrent region merging is *transitive*. Assume a program has three consecutive concurrent regions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$; $\mathcal{C}_1$ is mergable with $\mathcal{C}_2$, and $\mathcal{C}_2$ is mergable with $\mathcal{C}_3$. Consequently, $\mathcal{C}_1$ is also mergable with $\mathcal{C}_3$. Because there are no ordering guarantees between contiguous concurrent regions, the JIT compiler can reorder concurrent regions. Consider the following example, which shows three consecutive concurrent regions:



In the example above, $\mathcal{C}_1$ and $\mathcal{C}_3$ are executed by the same thread $\mathcal{T}_1$ and $\mathcal{C}_2$ is executed by a separate thread $\mathcal{T}_2$. The arrows indicate the program order in which the concurrent regions are defined in the source code. Since the timing of a thread executing a concurrent region is non-deterministic, the JIT compiler can assume that $\mathcal{T}_1$ finishes executing $\mathcal{C}_1$ and $\mathcal{C}_3$ before $\mathcal{T}_2$ executes $\mathcal{C}_2$. The timing assumptions cannot introduce new behaviors but fix a certain thread schedule. As a result, reordering $\mathcal{C}_2$ and $\mathcal{C}_3$ is a legal code transformation, since the reordering results in exactly the same behavior as the fixed thread schedule. Concurrent region merging is *symmetric*. If a concurrent region $\mathcal{C}_1$ is mergable with $\mathcal{C}_2$, $\mathcal{C}_2$ is also mergable $\mathcal{C}_1$.

6.1.2 Non-contiguous concurrent region merging

Two concurrent regions $\mathcal{C}_1$ and $\mathcal{C}_2$ are *non-contiguous* if there is at least one instruction i between the `cEnd` CIR-node of $\mathcal{C}_1$ and the `cStart` CIR-node of $\mathcal{C}_2$. If there exists a legal partial order of concurrent and sequential regions (that contain no synchronization action) that allows to reorder a concurrent region with a sequential region, two non-contiguous concurrent regions can be transformed into two contiguous concurrent regions. Consider the example in Figure 6.4(a), which contains two non-contiguous concurrent regions:

```
1   call A
2   cStart;    //cRegion1
3   cBarrier
4     call B
5   cBarrier;
6   cEnd       //cRegion1
7   call C
8   cStart;    //cRegion2
9   cBarrier
10    call D
11  cBarrier;
12  cEnd       //cRegion2
```

(a) Original program.

```
1   call A
2   call C
3   cStart;    //cRegion1
4   cBarrier
5     call B
6   cBarrier
7
8
9
10    call D
11  cBarrier;
12  cEnd       //cRegion1
```

(b) After concurrent region merging.

Figure 6.4: Non-contiguous concurrent region merging.

The two concurrent regions $\mathcal{C}_1$ (line 2 - line 6) and $\mathcal{C}_2$ (line 8 - line 12) are non-contiguous, since the call to method `C()` is between $\mathcal{C}_1$ and $\mathcal{C}_2$. Assume that conditions $1-5$ hold for the sequential region that is defined in line 7. The legal partial orders of the function calls in Figure 6.4(a) are:

$$\begin{tikzcd}
 & B() & \\
A() \arrow[ur] \arrow[dr] & & \\
 & C() \arrow[r] & D()
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & B() \\
A() \arrow[r] & C() \arrow[ur] \arrow[dr] & \\
 & & D()
\end{tikzcd}$$

In every legal partial order of the example shown in Figure 6.4, function `A()` must return before function `B()`, `C()`, and `D()` can be executed. Furthermore, function `C()` must return before function `D()` can be called. However, since there is no ordering between the calls to the functions `B()` and `C()`, as well as `B()` and `D()`, the call to `C()` can be reordered with $\mathcal{C}_1$. As a result, $\mathcal{C}_1$ and $\mathcal{C}_2$ become contiguous concurrent regions and are therefore mergable. Figure 6.4(b) shows the result of merging the two concurrent regions into one concurrent region.

6.1.3 Parallel code execution optimization algorithm

This section describes a parallel code execution optimization (PCEO) algorithm that aims at finding a good setting to execute concurrent regions efficiently. The PCEO algorithm determines (i) if a concurrent region is executed sequentially, (ii) if a concurrent region is executed in parallel, and (iii) if and how concurrent regions are merged. The PCEO algorithm can coarsen the parallel code granularity by collapsing two or more concurrent regions into a single concurrent region. The PCEO algorithm uses profile information collected at run-time to determine the setting of the concurrent regions at method-level granularity. Note that the PCEO algorithm cannot provide a parallel code granularity that is finer than the parallel code granularity provided by the programmer. Figure 6.5 shows the PCEO algorithm in detail.

The PCEO algorithm takes the CIR, the profile information, and the merging function as an input. The profile information contains the execution time of each concurrent region that is contained in the method as well as the number of parallel concurrent region executions that are issued by the method. Note that a concurrent region is profiled if the concurrent region is executed sequentially but also if the concurrent region is merged with another concurrent region. I.e., if two concurrent regions are merged, the execution time of the two individual concurrent regions is profiled. Such a fine-grained performance profile allows the JIT compiler to adapt the merging configuration based on individual changes of the execution time of a particular concurrent region.

The PCEO algorithm uses two thresholds: the serialization threshold (T_SER) and the merging function. T_SER determines if a (merged) concurrent region is executed sequentially or in parallel. T_SER refers to the execution time of a concurrent region in cycles. Section 7.4 and Section 7.5 discuss how we determine the thresholds for a particular hardware configuration. The merging function is discussed later in this section.

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. As a result, parallelism is potentially oversubscribed. Similarly, an oversubscription factor smaller than 1 means that parallelism is undersubscribed.

The PCEO algorithm is an iterative work list algorithm. The algorithm completes if it finds a recompilation condition or if the work list is empty. From line 9 -

```
Input: CIR, inf (profile information), mf (merge function)
Output: CIR

int OS = inf.cRegionInvocations / numProcessors;
Worklist workList = inf.getRegionProfiles();
inf.recompile = false;
while (!inf.recompile || !workList.isEmpty()) {
        CRegion r = worklist.get();
        if (!r.hasMergeCandidates() {
                if (r.execTime < T_SER) {
                        if (r.isParallel) {
                                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;
                        for (Cregion c : mergeCandidates) {
                                worklist.remove(c);
                                r.mergedWith.add(c);
                                c.mergedWith.clear();
                                c.isParallel = false;
                                double mergedExecTime = r.execTime;
                                for (CRegion t : r.mergedWith) {
                                        mergedExecTime += t.execTime;
                                }
                                if (mergedExecTime > T_SER) {
                                        r.isParallel = true;
                                        inf.recompile = true;
                                        break;
                                }
                        }
                } else if (demerge) {
                        r.mergeWith.clear();
                        inf.recompile = true;
                } else {
                        worklist.removeAll(r.mergedWith);
                }
  }
  handleRemainingRegions();
```

Figure 6.5: Parallel code execution optimization algorithm.

line 20, the algorithm handles concurrent regions that have no merging candidates. More specifically, the algorithm checks from line 10 - line 15 if the execution time of the concurrent region is smaller than the serialization threshold. 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). From line 15 – line 19 the algorithm checks if the execution time of the concurrent region is larger than the serialization threshold. If the concurrent region is compiled to sequential code, the PCEO algorithm issues a parallelizing recompilation.

Line 20 – line 47 handle concurrent region merging and demerging. The functions `shouldMerge()` and `shouldDemerge()` return a boolean value that indicates if a concurrent region should be merged or demerged. Both functions are shown in Figure 6.6 and Figure 6.7, respectively. If the algorithm decides to merge a concurrent region with another concurrent region, the algorithm first removes all concurrent regions from the list of potential merging candidates that are already merged with the concurrent region (line 24). 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 26 – line 40, the algorithm iterates over the merging candidates. For each inspected 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 32 – line 34 compute the execution time of the merged concurrent region. If the merged execution time is larger than the serialization threshold, the merged concurrent region will be compiled to parallel code (line 36 and line 37).

Line 41 – line 44 handle the demerging. Demerging is currently done by recompiling all merged regions to either sequential or parallel code. Line 43 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 48. This function simply checks for each non-merged concurrent region if the execution time is above or below the serialization threshold and sets the `isParallel` flag accordingly.

Figure 6.6 shows the implementation of the `performMerge` function.

`performMerge()` returns a boolean value that indicates whether the concurrent region should be merged with other concurrent regions. The first check in line 2 examines the execution time of the concurrent region. If the execution time is below the serialization threshold, `performMerge()` returns true. Otherwise, line 5 computes the merge function (see later this section). The returned value (`boundary`) 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 a minor effect. Furthermore, if the current oversubscription factor is larger than the computed merge function value, `performMerge()` returns true in line 7.

```
 1 performMerge(ProfileInformation inf, double OS) {
 2         if (inf.execTime < T_SER) {
 3                 return true;
 4         }
 5         double boundary = computeMergeFunction(inf);
 6         if (!withinTolerance(boundary, OS) && OS > boundary) {
 7                 return true;
 8         }
 9         return false;
10 }
```

Figure 6.6: Algorithm that determines if a concurrent region should be merged.

Figure 6.7 shows the implementation of the `performDemerge()` function.

```
1 performDemerge(ProfileInformation inf, double OS) {
2         double boundary = computeMergeFunction(inf);
3         if (!withinTolerance(boundary, OS) && OS < boundary) {
4                 return true;
5         }
6         return false;
7 }
```

Figure 6.7: Algorithm that determines if a concurrent region should be demerged.

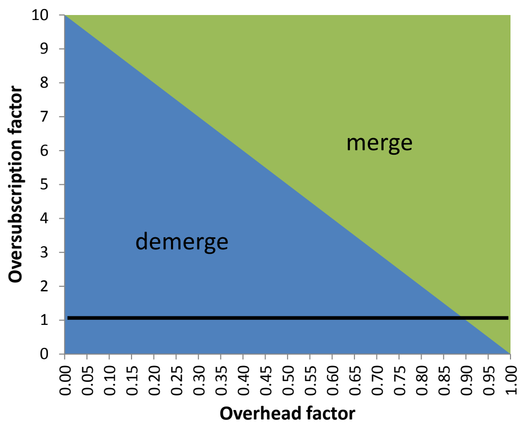
`performDemerge()` returns a boolean value that indicates whether the current set of concurrent regions should be demerged. Line 2 computes the merge function. Similar to the `performMerge()` function, line 3 checks if the current value of the OS is +/- 10% larger/smaller than the computed merge function value (`boundary`). In addition, if the current oversubscription factor is smaller than `boundary`, the `performDemerge()` function returns true in line 4.

The merge function defines if concurrent region merging or demerging is applied. The merge function can be supplied at Java VM startup. If no merge function is provided, the Java VM uses the default merge function that is described in Section 7.5.1. Figure 6.8 shows two sample merge functions.

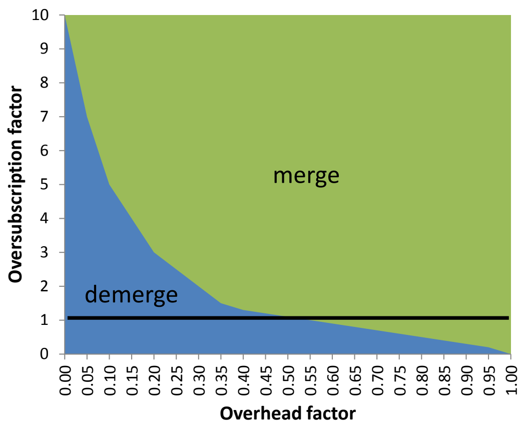
The x-axes in Figure 6.8 show the *overhead factor* (OH), which is defined as as follows:

$$OH = \frac{T\_SER}{execTime}$$

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



(a) Merge function 1



(b) Merge function 2

Figure 6.8: Merge function example.

merged with another concurrent region, or the concurrent region is serialized. The y-axes illustrate the oversubscription factor (OS), which is the number of generated tasks per available cores. An oversubscription factor of 1 means that the Java VM generates 1 task for each available core. The Java VM 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 Java VM-user 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. The merge functions illustrated in Figure 6.8 differ significantly with respect to efficiency. Merge function 1 aims at keeping the system highly balanced by striving for a high oversubscription factor even for high overhead factors. Such a merge function can result in a better peak performance (compared to merge function 2) due to better load balance. However, the efficiency can be lower, since a small speedup can require the usage of significantly more hardware resources. On the other hand, merge function 2 aims at a high efficiency. For example, an overhead factor of 0.5 yields an oversubscription factor of 1. For an overhead factor larger than 0.5, the merge function instructs the Java VM to generate less tasks than available processors.

The merge function can be provided to the Java VM with the following argument: `-X:merge_function="merge_string"`. For example, the merge function as illustrated in Figure 6.8(a) has the following merge string: `OH * -10 + 10`.

6.2 Parallel overhead reduction optimizations

This section illustrates two compiler optimizations that aim at providing an efficient implementation of the parallel version of a concurrent region. Section 6.2.1 shows an optimization that reduces the overhead from initializing a concurrent region in the parent thread. Section 6.2.2 presents an optimization that reduces the overhead for initializing merged concurrent regions. This optimization affects the parent thread as well as the child thread.

6.2.1 Concurrent region initialization optimization

Figure 6.9(a) shows an example that contains a concurrent call `cCall()`. The concurrent call (line 3) is not static and has one integer parameter that is loaded from field `o.f`. Figure 6.9(b) shows the CIR snippet that illustrates how the parameters of the concurrent call are passed to the concurrent region.

```
1 void foo(int x) {
2    ...
3    cCall(o.y);
4    sync();
5    o.f = ...;
6    return;
7 }
```

(a) Parallel overhead.

```
1 r0 = aload_0;
2 r1 = getfield;
3 cStart;
4 cBarrier;
5 call cCall(r0, r1);
6 cBarrier;
7 cEnd
```

(b) CIR of `foo()`.

```
1 r0 = aload_0;
2 cStart;
3 r1 = getfield;
4 cBarrier;
5 call cCall(r0, r1);
6 cBarrier;
7 cEnd
```

(c) CIR of `foo()` after optimization.

Figure 6.9: Parallel overhead reduction optimization example.

Since `cCall` is not static, it expects two parameters: The first parameter is a reference to the current object (`this` pointer). The `aload_0` IR-node stores the reference in the `this` pointer to `r0`. The second argument to `cCall` is the value of `o.f`. The `getfield` IR-node loads the value from `o.f` and stores the result in `r1`.

Note that the parent thread does not access `o.f` after the concurrent call is issued in line 3 up to the next `sync` method in line 4. The parent thread writes to `o.f` after the synchronization statement in line 5. As a result, the `getfield` bytecode is concurrency invariant to the concurrent region and can be moved into the concurrent region. Figure 6.9(c) illustrates the CIR snipped after the `getfield` IR node is moved into the concurrent region. Note that `aload_0` loads a reference from the local variable array and is therefore not concurrency invariant.

By moving the `getfield` IR-node into the concurrent region, the parent thread saves two memory operations: First, the parent thread does not perform the `get-field` operation. Second, the parent thread does not need to save the loaded value of `o.y` to the argument save area of the child thread (see Section 5.2). This code transformation has no impact on the parallel overhead of the concurrent region, since the child thread performs the `getfield` operation instead of the load from the argument save area.

6.2.2 Merged concurrent region initialization optimization

Consider the example in Figure 6.10(a), which contains two concurrent calls that can be merged into a single concurrent region. The two concurrent calls are non-static and take one local variable as parameter. Figure 6.10(b) shows the HIR of method `foo()` after concurrent region expansion and Figure 6.10(c) shows the concurrent stub that issues the merged concurrent region. Figure 6.10(b) contains one redundant store in line 6. The value of the `this` pointer cannot change between the store to the argument save area in line 3 and the store in line 6, which is marked with (*). As a result, the store in line 6 as well as the corresponding load from the argument save area in line 5 of Figure 6.10(c) are redundant and can be removed by the JIT compiler.

```
void foo(int x) {
  cCall(x);
  cCall(x+1);
  sync();
  return;
}
```

(a) Parallel overhead.

```
1   foo:
2     o = get_task_obj();
3     o.r[0] = this;
4     o.i[0] = x;
5     ...
6 * o.r[1] = this;
7     o.i[1] = x + 1;
```

(b) HIR of `foo()`.

```
1   cStub:
2     this = o.r[0];
3     r0 = i[0];
4     call __c_foo(this, r0);
5 * this = o.r[1];
6     r1 = o.i[1];
7     call __c_foo(this, r1);
```

(c) Concurrent stub.

Figure 6.10: Parallel overhead reduction optimization example.

7 Evaluation

This chapter presents the evaluation of the software platform as described in the previous chapters. Section 7.1 presents the system configuration of the two systems that we use for the evaluation. Section 7.2 evaluates the overhead of profiling concurrent regions with the instrumenting profiler. Section 7.3 presents the evaluation of the parallel overhead that is associated with every parallel execution and compares the parallel overhead of standard Java against our system. Section 7.4 represents an empirical methodology that enables a precise determination of the serialization threshold that is used in the parallel code execution optimization (PCEO) algorithm. Section 7.5 presents the evaluation of the Java Grande benchmarks [84]. Finally, Section 7.6 presents the evaluation of the dynamic compilation overhead that is introduced by dynamically parallelizing, serializing, and merging concurrent regions.

7.1 Sytem configuration

Table 7.1 shows the configuration of the two systems that we use to evaluate our automated parallel code optimization system. To illustrate that our system provides performance portability across a diverse set of hardware platforms, the two systems have a different number of cores (8 and 32 cores) as well as a different Linux kernel (32-bit and 64-bit).

The first system is equipped with 8 physical cores. We will thus refer to this configuration as the *8-core system* throughout the evaluation. All cores of the 8-core system run at 3 GHz; the system has 6 MB last-level cache and 8 GB of main memory. The second system is equipped with 32 physical cores and we will therefore refer to this system as the *32-core system.* All physical cores run at 2.13 GHz; the system has 24 MB L3 cache, which is shared between all cores on a processor. The system has 64 GB of main memory.

All experimental results are obtained by running the benchmarks on the modified version of the Jikes Research Virtual Machine (Jikes RVM) 3.1.2 in the `FullAdaptiveCopyMS` configuration. This configuration is similar to the `production` configuration, which uses the Immix [18] garbage collector. We cannot use the `production` configuration, since the Immix garbage collector is unstable. However, the

	8-core system	*32-core system*
Physical processors	2	4
Cores per processor	4	8
Model name	Intel Xeon E5450	Intel Xeon E7-4830
CPU frequency	3 GHz	2.13 GHz
L3 cache	6 MB	24 MB
Main memory	8 GB	64 GB
Operating system	Ubuntu 10.04.4	Ubuntu 11.10
Linux kernel	2.6.32. 32-bit	3.0.0, 64-bit

Table 7.1: Benchmark platform description.

`FullAdaptiveCopyMS` configuration uses the adaptive optimization system like the `production` configuration. The Java Virtual Machine (Java VM)-internal thread pool is taken from the `java.util.concurrent` package. All presented benchmark results use the same thread pool implementation. Table 7.2 summarizes the Java platform configuration.

	8-core system	*32-core system*
Java version	1.6.0_17	1.6.0_24
Available processors	8	32
Minimum heap size	2,000 MB	2,000 MB
Maximum heap size	2,000 MB	2,000 MB
GC threads	4	4

Table 7.2: Java platform configuration.

The 8-core system uses Java version 1.6.0_17 and the 32-core system uses Java version 1.6.0_24. The number of available cores is set equal to the number of physical cores of the corresponding platform. The number of available processors is accessible to the application by calling the `Runtime.getRuntine().availableProcessors()` function. The minimum heap size and the maximum heap size is set to 2,000 MB for both systems. We use this configuration, since it provides the most stable behavior of the Jikes RVM. Finally, the number of GC threads is set to 4 for both systems. We use a parallel collector to reduce the overhead from garbage collection. However, increasing the number of GC threads higher than 4 results in an unstable behavior of the Jikes RVM.

7.2 Profiling overhead

This section presents the evaluation of the overhead that is introduced by profiling concurrent regions. The overhead of profiling the execution time of concurrent regions has three main sources: (i) measuring the execution time of a concurrent region, (ii) storing the measured time (sample) into the thread-local database, and (iii) committing samples from the thread-local database to the global database. The execution time of a concurrent region is measured using the `rdtsc` (read time stamp counter) instruction. Since the `rdtsc` instruction is executed only twice for each concurrent region measurement and executing the `rdtsc` instruction is extremely cheap, the overhead from measuring the execution time is negligible small. Storing a sample into the thread-local database incurs a running time overhead from looking up the current sample and adding the current execution time to the existing samples. Note that storing a sample into the thread local database does not require synchronization with other threads. However, committing thread-local samples to the global database is guarded by a lock to ensure mutual exclusion. Mutual exclusion is required, since multiple threads write to and read from shared data. As a result, updates to the global sample database can become a performance bottleneck if numerous threads perform updates concurrently. To reduce the number of updates to the global database, samples are only committed if a thread has collected a certain amount of samples. In addition, the average of the current samples must differ by more than 30% compared to the last committed average (see Section 5.3.2).

Recording the execution time of every concurrent region execution results in a significant performance penalty if the execution time of the concurrent region is small, i.e., in the order of the time it takes to store a sample into the thread-local database. For this reason, the samples are stored into the thread-local database at a predefined *record interval*. As such, the record interval determines the frequency at which samples are recorded. The advantage of a short record interval is that samples are recorded at a high frequency. This results in a better precision compared to a longer record interval. The disadvantage, however, is that the running time overhead of a short record interval is higher than for that of a large record interval.

The profiling overhead of counting the number of concurrent region invocations has two sources: (i) incrementing a counter and (ii) storing the counter value to the thread-local database at the end of the method. The subsequent results only consider the overhead from execution time profiling. However, the performance results in Section 7.5 contain the overhead from profiling the execution time of concurrent regions as well as the overhead from counting the number of concurrent region invocations.

7.2.1 Experimental methodology

The running time overhead of execution time profiling is determined by using a synthetic benchmark. The synthetic benchmark consists of a simple loop that contains one concurrent region that in turn contains one method call. The loop body is executed 10^7 times to get a stable execution execution time of the benchmark. Since each loop iteration contains one concurrent region, the Java VM can potentially profile 10^7 concurrent region executions.

The baseline for the profiling overhead evaluation is the execution time of the un-profiled version of the synthetic benchmark that is compiled to sequential code. Compiling the concurrent region to sequential code is better suited to measure the profiling overhead, since sequential code does not incur a parallel overhead that is not constant as we show later in this chapter. 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. More specifically, values for the record interval range from 0 cycles (the running time of every concurrent region is recorded) 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 from committing a thread-local samples to the global database.

The profiling overhead of concurrent regions that are compiled to parallel code is potentially larger, since each thread updates thread-local samples to a global database and updates to the global database are protected by a critical section. To measure the overhead of multiple threads performing updates to the global profiling database *without* including the parallel overhead, the parallel version of the synthetic benchmark starts regular Java threads that execute the same concurrent region that is compiled to sequential code. To ensure that all threads execute the loop that contains the concurrent region at the same time (this causes the maximum contention at the global database) the synthetic benchmark has a barrier before the main benchmark loop.

The execution times presented in Section 7.2.2 are obtained by executing the synthetic benchmark 20 times in the same Java VM instance. The first 10 runs are warm-up runs [35] and excluded from the performance evaluation. The arithmetic average over the last 10 runs are presented in Section 7.2.2. The error bars in Figure 7.1 show the standard deviation over the ten performance runs.

7.2.2 Results

Profiling overhead

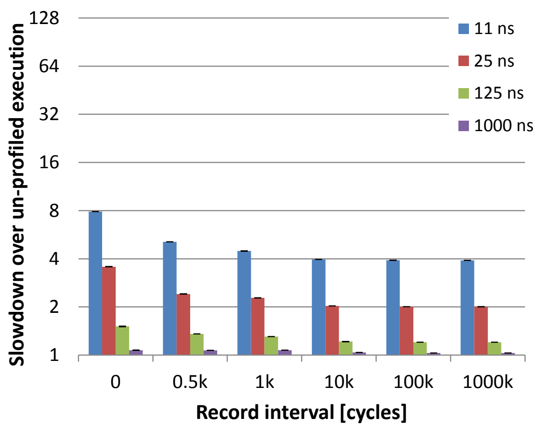
Figure 7.1 shows the profiling overhead for a combination of record intervals, concurrent region execution times, and number of threads. The x-axes contain the record intervals in cycles. A value of 0 on the x-axis means that the execution time of every concurrent region execution is recorded. A value of 1,000 on the x-axis means that at least 1,000 cycles must have passed since the last recorded sample so that a new sample is stored to the thread-local database. The y-axes show the slowdown factor of the profiled version over an un-profiled execution. The slowdown factor is calculated by divided the execution time of the profiled version by the execution time of the un-profiled version. Figure 7.1 shows the profiling overhead for concurrent region execution times of 11 ns, 25 ns, 125 ns, and 1,000 ns. The profiling overhead is shown for 1, 2, 8, 16, and 32 parallel threads.

Figure 7.1(a) illustrates the profiling overhead of a single-threaded execution. The profiling overhead using one thread significantly depends on the record interval as well as on the execution time of the concurrent region. The shortest concurrent region execution time (11 ns) and a sampling interval of 0 results in the highest profiling overhead and a slowdown of 8X. A slowdown of 8X is the maximum possible slowdown using 1 thread, since the concurrent region contains a call to an empty method body. As a result, the execution time of the un-profiled concurrent region consists of one call instruction and one return instruction. As explained in Chapter 5, the just-in-time compiler (JIT compiler) can inline methods into a concurrent region and instruments a concurrent region only if the concurrent region either contains a method call or a loop. Since the method body is empty, the JIT compiler would compile the concurrent region to un-profiled, sequential code. We adapted the JIT compiler to instrument any concurrent region to determine the theoretical maximal possible overhead. If the execution time of the concurrent region is 25 ns, the profiling overhead drops to 3.6X. A concurrent region execution time of 1,000 ns causes a profiling overhead of 7% only.

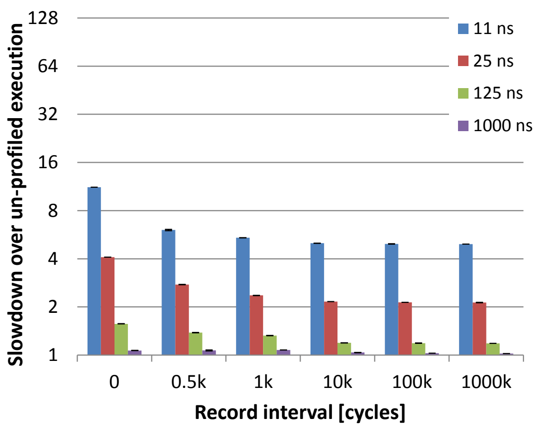
Increasing the record interval also causes a degradation of the profiling overhead. For example, the profiling overhead of a concurrent region execution time of 11 ns at a record interval of 100,000 cycles is 4X. Further increasing the record interval to 1,000,000 cycles also causes a slowdown of 4X, which corresponds to an absolute performance penalty of 44 cycles. Since increasing the record interval does not reduce the profiling overhead, the profile overhead comes from executing the `rdtsc` instruction and checking whether the sample should be put into the thread-local database. Storing a sample into the thread-local database does not introduce a measurable performance penalty at a record interval larger than 10,000 cycles although a lower record interval causes more samples to be stored, as shown in Figure 7.2. The reason is that stores to the thread-local database are executed infrequently.

Longer concurrent region execution times result in a lower profiling overhead, since the profiling overhead is constant for a single-threaded execution. For example, a concurrent region execution time of 25 ns causes a profiling overhead of 4X for a record interval of 0 cycles. The profile overhead drops significantly with an increase of the concurrent region execution time: 125 ns causes a profiling overhead of only 1.5X and 1,000 ns slows the application down by 1.07X. The slowdown for the longest concurrent region execution time (1,000 ns) at a recording interval of 0 cycles is only 1.33X. The longest concurrent region execution time causes a profiling overhead of only 1.03 (3%) at record intervals of 100,000 cycles and 1,000,000 cycles.

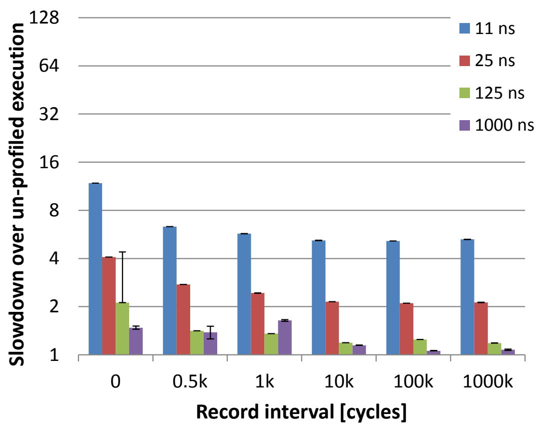
Figure 7.1(b) shows the profiling overhead using two parallel threads. The profiling overhead using two threads is similar to using a single thread. The largest



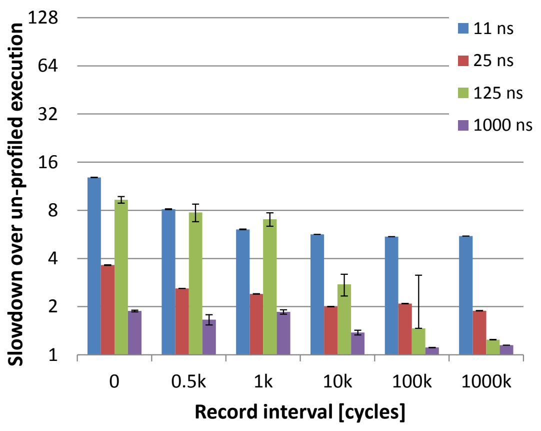
(a) Profiling overhead 1 Thread.



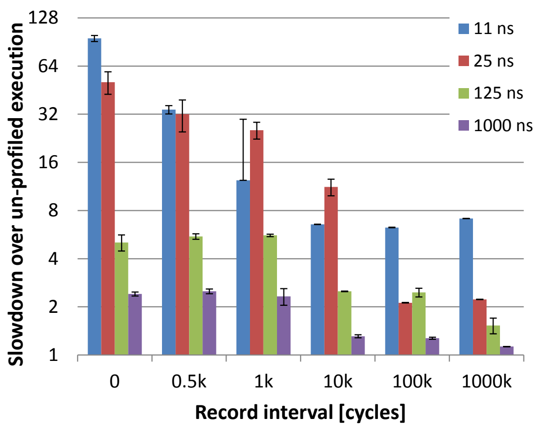
(b) Profiling overhead 2 Thread.



(c) Profiling overhead 8 Threads.



(d) Profiling overhead 16 Threads.



(e) Profiling overhead 32 Threads.

Figure 7.1: Profiling overhead results.

difference between a two-threaded execution and a single-threaded execution appears with a concurrent region execution time of 11 ns and a record interval of 0 cycles. While a single-threaded execution causes a slowdown of 8X, using two threads causes a slowdown of 11X. The reason is that a record interval of 0 cycles stores every taken sample into the thread-local database. Committing samples to the global database can be a performance bottleneck due to locking. A concurrent region execution time of 25 ns causes a slowdown of 4X (for two threads) compared to 3.6X using one thread. The number of committed samples drops significantly (see Figure 7.2 and Figure 7.3) and therefore incurs less frequent commits to the global database. The profiling overhead of a concurrent region execution of 1,000 ns using two threads is equal to the profiling overhead using one thread in the same configuration. Similar to the single-threaded execution, increasing the record interval decreases the profiling overhead. For example, a record interval of larger than 10,000 cycles and a concurrent region execution time that is larger than 100 ns results in no measurable difference in the profiling overhead of the two-threaded and the single-threaded execution.

Figure 7.1(c) shows the profiling overhead using eight parallel threads. Similar to using two threads, the maximum increase of profiling overhead compared to using a smaller number of threads is incurred by a record interval of 0 cycles and a concurrent region execution time of 11 ns. This configuration causes a slowdown of 12X over the un-profiled version. In contrast to the two-threaded benchmark execution, a concurrent region execution time of 1,000 ns causes more profiling overhead (1.48X). This overhead is caused by committing samples to the global database. Increasing the record interval causes a significant decrease of the profiling overhead with one exception: A concurrent region execution time of 1,000 ns at a record interval of 1 million cycles (1.64X) causes more profiling overhead than the profiling overhead at a record interval of 500,000 cycles (1.38X). Such a behavior can be explained by the small difference in the record interval (2X) and that this configuration is unlucky for performance of locking. Furthermore, the standard deviation for a record interval of 500,000 cycles and 1,000 ns is relatively high (1.12X). A record interval of 100,000 does not cause more profiling overhead irrespective of the concurrent region execution time.

Figure 7.1(d) shows the profiling overhead using 16 parallel threads. The profiling overhead using 16 threads significantly differs from the profiling overhead using one or two threads for a concurrent region execution time of 125 ns: A longer concurrent region execution time (125 ns) causes more profiling overhead than a shorter concurrent region execution time (25 ns). The reason is that the profiling overhead is not constant with respect to the number of threads due to committing samples to the global database. Figure 7.3(d) shows that the number of samples committed to the global database is higher (20X for a record interval of 0, 500, and 1,000 cycles). Recall that samples are only committed to the global database if the average of the last n collected samples (n increases from 0 up to 50 in steps of 10) is +/- 30% of the last committed average. Since the number of committed samples is large for a concurrent region execution time of 125 ns the execution time varies significantly. The reason for the large variation in the execution time can either be the barrier at which all threads wait prior to entering the benchmark loop or Java VM services such as memory allocation and/or garbage collection. The difference of the profiling overhead of 25 ns and 125 ns compared to using a smaller number of threads is smaller at a record interval of 10,000 cycles. Comparing Figure 7.3(c) and Figure 7.3(d) reveals that that there are 3X less committed samples in Figure 7.3(c). A record interval of 100,000 cycles and more shows similar behavior as the single-threaded and two-threaded execution. However, the lowest profiling overhead for a concurrent region execution time of 1 million is around 11% to 15% for a record interval of 100,000 cycles and 1 million cycles, respectively.

Figure 7.1(e) shows the profiling overhead using 32 parallel threads. Using 32 threads significantly increases the profiling overhead of short concurrent region execution times (i.e., 11 ns and 25 ns). For example, the slowdown for a concurrent region execution time of 11 ns at a record interval of 0 ns introduces a performance penalty of almost 100X. Increasing the record interval for short concurrent regions significantly reduces the record overhead for the same reasons as listed above. However, the record overhead of the longest concurrent region execution time remains unaffected by increasing the record interval. The reason is that the execution time of 1,000 ns is longer than the record interval. As a result, every sample is recorded into the thread-local database and is potentially committed to the global database. There is a significant reduction of the profiling overhead when increasing the record interval from 1,000 cycles to 10,000 cycles. As shown in Figure 7.2, the record rate (see later this section) drops from 100% to 25%. Consequently, the overhead from recording a sample in the thread-local database as well as the contention at the global database is lower. The lowest recording overhead for 32 threads is 13% for the longest concurrent region execution time and the largest record interval.

To summarize, the profiling overhead for programs that use one or two threads is determined by the overhead from (i) measuring the execution time and (ii) putting the sample into the thread-local database. An increasing number of threads increases the profiling overhead significantly if the record interval is short. However, if the execution time of the concurrent region is long (i.e., in the order of a microsecond) and the record interval is large (i.e., 1,000,000 cycles) the profile overhead is modest (13%). Note that the profiling overhead is measured over an un-profiled *sequential* execution. As shown in Section 7.3 the parallel overhead is several orders of magnitude larger compared to the profiling overhead. As a result, the profiling overhead is most relevant for concurrent regions that are compiled to sequential code.

To keep the profiling overhead for sequential code low, the record interval for sequential code can be larger than the record interval for profiling concurrent code without compromising the precision of the profile. I.e., if the execution time of a concurrent region is such that the concurrent region is compiled to sequential code, choosing a record interval that is significantly smaller than the serialization threshold (see Section 7.4) cannot trigger a recompilation, since the execution time is too short. For parallel code, however, the profiling overhead is in the noise of the parallel overhead (see Section 7.3). Another technique to reduce the profiling overhead is to change to record interval at run-time. For example, the record interval can change periodically to longer/shorter record intervals.

Samples committed to the thread-local database

Figure 7.2 shows the ratio of samples that are stored into the thread-local database.

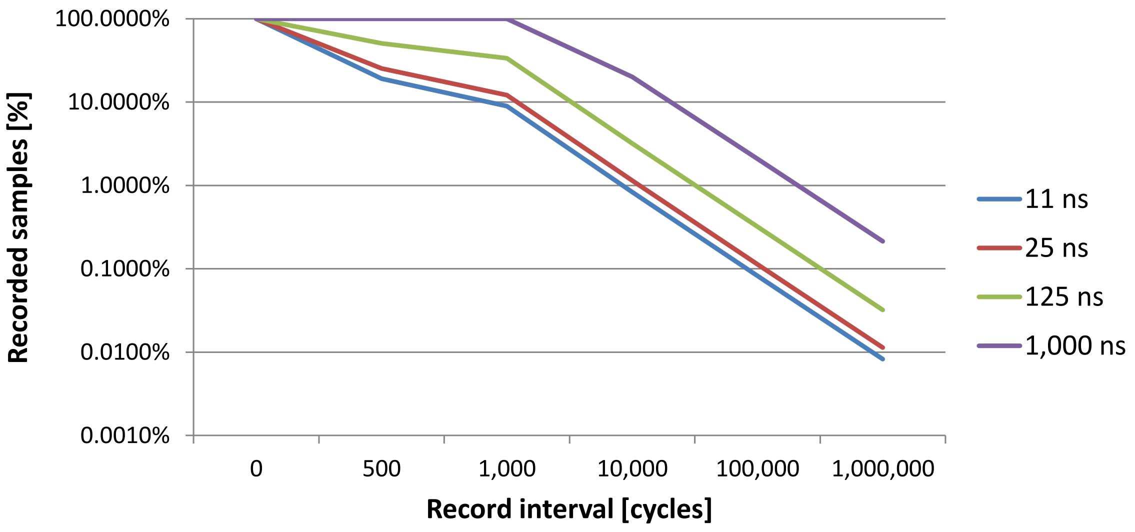


Figure 7.2: Ratio of recorded samples to the thread-local database.

Recall that the record interval determines if a sample is put into the thread-local database or not. The data from Figure 7.2 is obtained from executing the same synthetic benchmark as described above. The x-axis shows the record interval in cycles. The y-axis shows, on a logarithmic scale, the ratio of recorded samples. For example, 100% on the y-axis corresponds to recording all samples; 0% corresponds to recording no sample. A higher number on the y-axis corresponds to a higher precision of the profile. The data for the parallel versions are not presented since the record interval is identical for all threads.

Figure 7.2 reveals that the number of recorded samples depends on both, the concurrent region execution time and the record interval. For example, a record interval of 1,000 cycles causes the recording of 10% for a concurrent region execution time of 11 ns and 25 ns and the recording of all samples (100%) for a concurrent region execution time of 1,000 ns. Note that if the execution time of the concurrent region is longer than the record interval, the execution time of every concurrent region is recorded. That is why the percentage of recorded samples remains at 1 (1,000 ns) for a record interval of 500 cycles and 1,000 cycles. As a result, the number of recorded samples for longer concurrent region execution times is higher than for short concurrent region execution times.

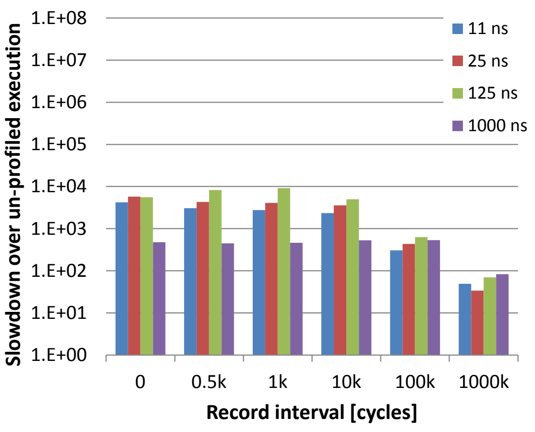
Samples committed to the global database

Figure 7.3 shows the total number of samples that are committed from the thread-local database to the global database for 1, 2, 8, 16, and 32 threads. The x-axes in Figure 7.3 show the record intervals in cycles. The y-axes show the total number of commits in a logarithmic scale. The number of committed samples is presented for the concurrent region execution times of 11 ns, 25 ns, 125 ns, and 1,000 ns.

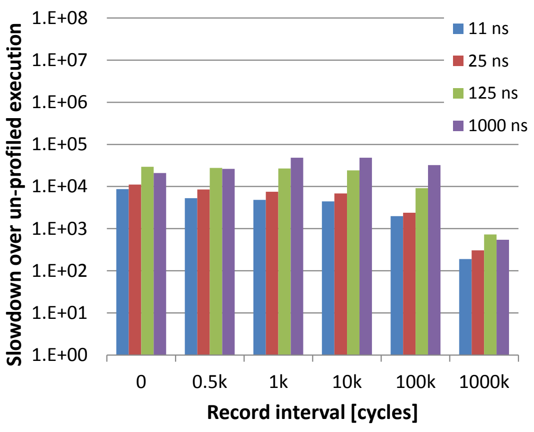
The number of committed samples depends on two factors: First, the record interval determines how many samples are recorded into the thread-local database. If the record interval is large, there are less samples recorded and consequently less samples committed to the global database. Second, if the execution time of a concurrent region is stable, i.e., the variation of the average of the execution time is less than 30%, the sample is not committed to the global database, since a stable execution time of a concurrent region does not require changes to the parallel code granularity.

Figure 7.3(a) shows the number of committed samples for the single-threaded execution. The single threaded execution commits the least samples, since there is only one thread. The record interval of 0 cycles incurs approximately the same number of commits for the concurrent region execution times of 11 ns, 25 ns, and 125 ns although each sample is recorded in the thread-local database. For example, the concurrent region execution time of 25 ns incurs 5500 commits of 125 ns incurs 5700 commits. The longest execution time of a concurrent region incurs only 477 commits. The reason is that longer execution times are less likely to incur a variation of more than 30% than shorter execution times. For short execution times, low-level performance characteristics like the cache hit/miss rate are more likely to incur a variation in the execution time compared to longer execution times. Increasing the record interval significantly reduces the number of commits. The number of commits drops below 100 for a record interval of 1 million cycles for every concurrent region execution time.

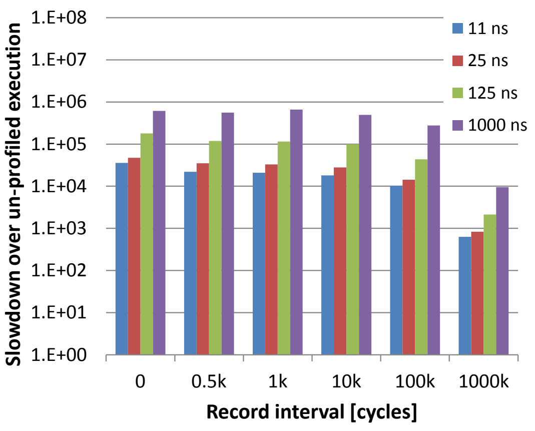
Figure 7.3(b) shows the number of committed samples for the two-threaded execution. The number of committed samples doubles for the concurrent region execution times of 11 ns and 25 ns at a record interval of 0 samples compared to the single-threaded execution. Such a behavior is expected when using two threads instead of one. However, the number of commits significantly increases for the concurrent region execution times of 125 ns and 1,000 ns. Compared to the sequential execution, the number of commits increases by a factor of 6 for 125 ns and a factor of 40 for a concurrent region execution time 1,000 ns. Determining the reason(s) for this behavior requires further investigation. While increasing the record interval causes less commits for the concurrent region execution times of 11 ns, 25 ns, and 125 ns, the number of commits increases for the 1,000 ns concurrent region execution



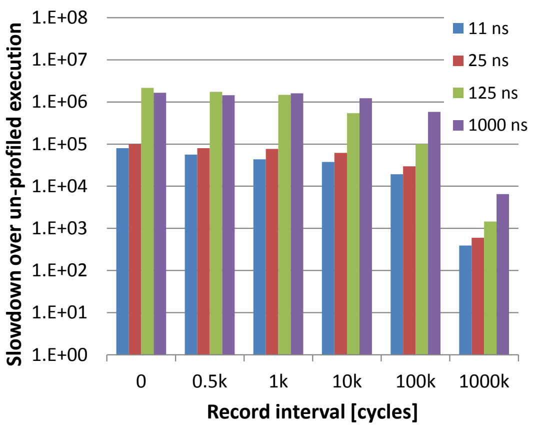
(a) Committed samples for 1 thread.



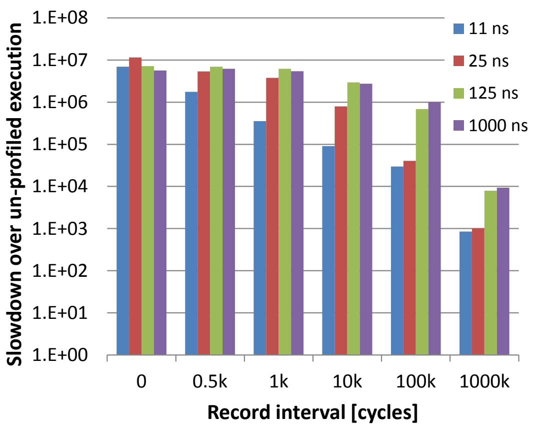
(b) Committed samples for 2 threads.



(c) Committed samples for 8 threads.



(d) Committed samples for 16 threads.



(e) Committed samples for 32 threads.

Figure 7.3: Number of samples committed to the global database.

time. A record interval of 0 cycles incurs 21,000 commits, 500 cycles incur 26,000 commits, 1,000 and 10,000 cycles incurs 48,000 commits. The reason for this behavior also requires further investigation. A record interval of 1,000,000 cycles incurs the least number of commits. The concurrent region execution time of 125 ns incurs the most commits (700), followed 500 commits for 1,000 ns.

Figure 7.3(c) shows the number of committed samples for an execution using 8 threads. Using 8 threads increases the number of committed samples significantly at a record interval of 0 cycles for a concurrent region execution time of 1,000 ns. Compared to using two threads, the number of commits increases by a factor of 30. Shorter concurrent region execution times increase by a factor of 4 (11 ns and 25 ns) and a factor of 6 for 125 ns. An explanation of this behavior also requires further investigation. Unlike in the two-threaded execution, increasing the record interval decreases the number of committed samples. The largest record interval causes 600 commits for a concurrent region execution time of 11 ns, 800 commits for 25 ns, 2,000 commits for 125 ns, and 10,000 commits for a concurrent region execution time of 1,000 ns. Committing 10,000 samples corresponds to a committing 1 of 8,000 measured execution times.

Figure 7.3(d) shows the number of committed samples for an execution using 16 threads. Using 16 threads has a similar characteristic as using 8 threads. While the number of committed samples increases by a factor of approximately 2 for a concurrent region execution time of 11 ns, 25 ns, and 1,000 ns at a record interval of 0 cycles there is a significant increase for a concurrent region execution time of 125 ns (factor 12). The record intervals 500 cycles and 1,000 cycles show similar behavior. For a record interval larger than 1,000 cycles the number of committed samples drops significantly, since the number of samples put into the thread-local database also drops significantly. The number of committed samples for the longest record interval is lower than using 8 threads. Since the number of committed samples depends on the variation of the execution time, there are more commits of the execution time is less constant.

Figure 7.3(e) shows the number of committed samples for an execution using 32 threads. In contrast to using 16 threads, the number of committed samples increases significantly for a concurrent region execution time of 11 ns and 25 ns (factor 80 and factor 115) and causes only a modest increase for the concurrent region execution times of 125 ns and 1,000 ns (factor of 3). Using 32 threads causes a large variation for the short concurrent region execution times. Increasing the record interval for a concurrent region execution time of 11 ns causes a decrease of the committed samples according to the reduced number of samples that are put into the thread-local database. Longer concurrent region execution times are less affected by increasing the record interval, since the number of samples recorded to the thread-local database drops slower. However, the total number of committed samples remains low for a record interval of 1 million cycles: A concurrent region execution time of 11 ns commits 800 samples, 25 ns commits 1,000 samples, 125 commits 8,000 samples, and 1,000 ns commits 9,000 samples.

7.3 Parallel overhead

This section presents the evaluation of the parallel overhead that is associated with every parallel execution for standard Java and our software platform. In general, the parallel overhead consists of three distinct sources: (i) infrastructure overhead, (ii) scheduling overhead, and (iii) synchronization overhead. The infrastructure overhead is composed of a collection of overheads that are introduced for parallel programs. For example, parallel code must be outlined to a separate function that can be called by a separate thread. Outlining introduces an overhead due to the function call. Furthermore, a parallel execution requires object creation, e.g., a new parallel task object. The overhead comes from allocating the object on the heap, initializing the object, and garbage collecting the object. The resulting overhead strongly depends on the implementation of the memory allocator and garbage collector in the Java VM. Note that compiler optimizations that aim at reducing the memory consumption of a program cannot be applied to parallel task objects, since parallel task objects escape to another thread.

The scheduling overhead results from creating a new thread or submitting the task to a thread pool. Submitting a parallel task to a thread pool incurs an overhead since an idle thread must be woken up or the task object must be put into a work queue for a later execution. If a thread pool is used, concurrent accesses to shared resources like a work queue can limit the scalability of the thread pool.

Finally, the application incurs a synchronization overhead for checking if a parallel task has finished execution. Note that Java requires a synchronization action to ensure the communication between two threads. Other potential performance bottlenecks for parallel programs such has accessing a shared lock or critical sections are not considered in this evaluation, since these performance bottlenecks are a characteristic of the application and independent of the parallel overheads that are discussed in this section.

7.3.1 Experimental methodology

To quantify the three different sources of parallel overhead, the infrastructure overhead, the scheduling overhead, and the synchronization overhead, a synthetic benchmark is used. The benchmark consists of executing 10 million tasks that contain lock free code. The synthetic benchmark is executed using four configurations. First, the baseline configuration is a *sequential* execution of the 10 million tasks *without* allocating 10 million task objects, i.e., one task object is executed 10 million times. The execution time of the baseline configuration T_b is the execution time of the synthetic benchmark with no parallel overhead. To determine the performance impact of the infrastructure overheads (e.g., memory allocation and garbage collection) the benchmark allocates 10 million task objects that are executed sequentially by a single thread. The infrastructure overhead is referred to as T_i.

The scheduling overhead, T_s, is determined using one thread to execute 10 million task objects. T_s is the time from the first object allocation until the last parallel task is processed. To get the time when the last task is processed without including the synchronization overhead, the last action of each task object writes the current time to a global field.

The synchronization overhead T_{sync} is the time from the completion of the last task until the producer of the tasks has checked that all tasks have been processed. The producer thread maintains a reference to all task objects and can check if a task completed by calling the `get()` method on the reference, which is of type `Future`.

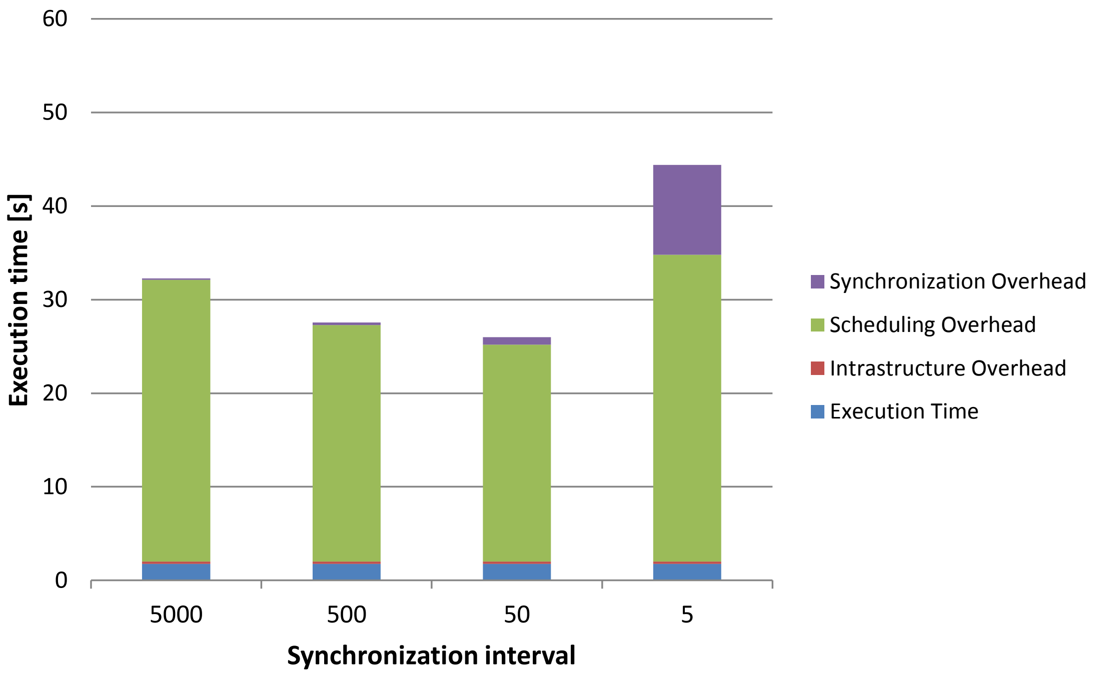
The total execution time of the parallel execution of the synthetic benchmark is: $T_{total} = T_b + T_i + T_s + T_{sync}$. The execution times that are presented in the next section are obtained from the 32-core system. The numbers are the averages over five runs in the same Java VM instance. Similar to the previous measurements, we measure the steady-state performance of the application. The parallel overhead of a pure Java-based synthetic benchmark is compared against a version of the synthetic benchmark that uses concurrent statements. The next section presents the detailed breakdown of the parallel overhead of both versions.

7.3.2 Results

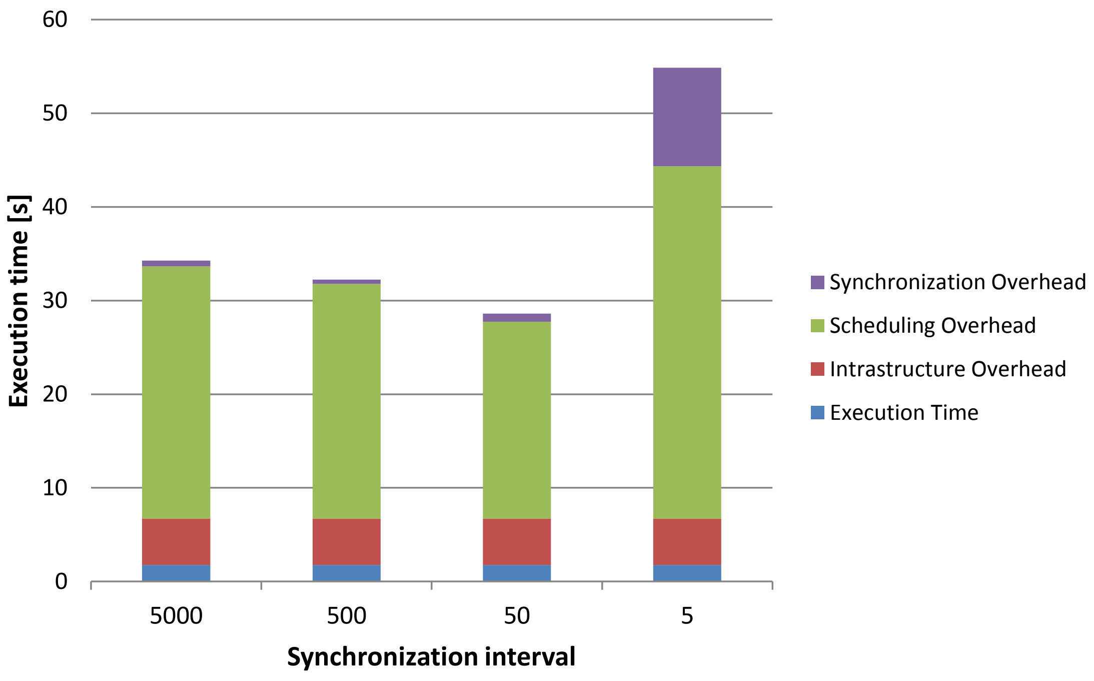
Figure 7.4 illustrates the breakdown of the parallel overhead. Figure 7.4(a) shows the parallel overhead of the Executor framework and Figure 7.4(b) shows the parallel overhead of a concurrent loop statement. The x-axes in Figure 7.4(a) and Figure 7.4(b) show the synchronization interval, i.e., the number of parallel task objects issued between two synchronization operation. Recall that a synchronization operations ensures that all submitted tasks have finished. For example, a value of 5,000 on the x-axis means that a synchronization operation is issued after 5,000 tasks have been submitted to the thread pool. The y-axes in Figure 7.4(a) and Figure 7.4(b) show the total execution time of the synthetic benchmark in microseconds.

Executor framework

The scheduling overhead contributes the largest part of the parallel overhead in Figure 7.4(a). For a synchronization interval of 5,000, scheduling contributes 97% of the total execution time. The scheduling overhead decreases with a decreasing synchronization interval until the synchronization interval is 50. The total execution of the synthetic benchmark decreases although the synchronization overhead increases, because the synchronization overhead increases only slightly. One reason for the reduction in the scheduling overhead is that the work queue in the thread pool is potentially smaller and that accessing the work queue causes a significant overhead. The maximum size of the work queue for a synchronization interval of 5,000 is 5,000 elements and for a synchronization interval of 50, the thread pool internal work queue must hold at most 50 elements. For a synchronization interval



(a) Parallel overhead of the Executor framework.



(b) Parallel overhead of our approach.

Figure 7.4: Breakdown of the parallel overheads.

of 5, the scheduling overhead increases significantly. The reasons for this behavior requires further investigation.

The synchronization overhead is the second largest source of parallel overhead. As illustrated in Figure 7.4(a) the synchronization overhead depends on the number of synchronization operations executed. For example, for a synchronization interval of 5,000, the synchronization overhead contributes only 0.5% of the total execution time. For a synchronization interval of 5, the synchronization overhead contributes 22% of the total execution time.

Finally, the infrastructure overhead contributes the least of the three sources of overhead. The infrastructure overhead does not depend on the number of synchronization operations and is therefore constant for all evaluated configurations in Figure 7.4(a). The infrastructure overhead contributes between 0.6% and 0.8% of the total execution time.

Concurrent `for` statement

Figure 7.4(b) shows the breakdown of the parallel overhead using a concurrent `for` statement. The performance data shown in Figure 7.4(b) is obtained *without* profiling concurrent regions and dynamically recompiling sequential/concurrent regions. Instead, each concurrent statement is compiled to parallel code. The results in Figure 7.4(b) are similar to the results shown in Figure 7.4(a) with the exception that infrastructure overhead is higher compared to the Java version. The reasons for the higher infrastructure overhead are that (i) the JIT compiler-generated parallel code contains additional method calls (e.g., to load the reference to the parallel task object) and (ii) that each task object contains a scratch area to which method arguments are saved. The size of the scratch area is currently a constant value (64 bytes). Optimizing the size of the scratch area of the Java VM-internal task objects reduces the memory footprint and also the infrastructure overhead. Note that the size of the scratch area can be determined at compile-time by the JIT compiler.

7.4 Determining the *serialization threshold*

The PCEO algorithm determines if a concurrent region is executed sequentially or in parallel. The PCEO algorithm uses a constant threshold that determines the optimization strategy: the *serialization threshold*. If the execution time of a concurrent region is below the serialization threshold and the concurrent region cannot be merged with another concurrent region, the JIT compiler compiles the concurrent region to sequential code. The serialization threshold determines the efficiency at which parallel code is executed. A low efficiency can imply the usage of all processors with a modest performance gain.

The following section describes one methodology to determine the serialization threshold so that the serialization threshold determines the point at which the parallel overhead is equally large as the performance gain from a parallel execution. I.e., the overhead that is introduced by parallelism is equally large as the execution time of the parallel task. This standard serialization threshold is determined at deployment of the Java VM and reported to the user. Based on the reported value and in combination with the merging function, the user of the Java VM can configure the efficiency at which the parallel application is executed. The corresponding code generation is done fully automatically by the Java VM.

7.4.1 Experimental methodology

We use a synthetic benchmark to determine the serialization threshold as described above. The synthetic benchmark consists of 900,000 work items that can be assigned to task objects for a parallel execution. The synthetic benchmark contains one concurrent loop. The Java VM is configured such that no concurrent region merging is applied. The synthetic benchmark can produce different task sizes. For example, if all 900,000 work items are assigned to a single task object, the synthetic benchmark produces the largest parallel task granularity (900,000). If each of the 900,000 work item is assigned an individual task object, the synthetic benchmark incurs the smallest parallel task granularity (1).

To determine a good serialization threshold, i.e., code is not serialized when a parallel execution yields a performance gain and a parallel execution is not slower than a sequential execution, we use an empirical evaluation. More specifically, the serialization threshold is set to 0 cycles, 10,000 cycles, 50,000 cycles, and 100,000 cycles. A serialization threshold of 0 cycles corresponds to an execution in which every concurrent region executes in parallel. Analogously, a serialization threshold of 100,000 cycles corresponds to an execution in which a concurrent region executes in parallel only if the execution time of the concurrent region is longer than 100,000 cycles. Using the four different values for the serialization threshold, we measure the execution time of the synthetic benchmark for different parallel task granularities.

The performance results that are presented in the next section are the arithmetic average over 10 executions in the same Java VM instance. The record interval is set to 1 million cycles. We use the 32-core system with 32 threads to obtain the performance data. Using a smaller number of threads gives similar performance results, since the scheduling overhead is largely independent of the number of threads. The sequential execution time of the benchmark is 993 ms.

7.4.2 Results

Figure 7.5 shows the relative performance of the parallel execution compared to the sequential execution for a decreasing parallel code granularity. The x-axis illustrates the parallel code granularity. As shown in Figure 7.5, the performance of the four versions is similar for a parallel code granularity larger than 90. The reason is that the execution time of the concurrent region with the chosen parallel code granularity is larger than 100,000 cycles. As a result, the concurrent regions are executed in parallel.

At a parallel code granularity of 18, the 100,000 cycles serialization threshold version issues a recompilation that serializes the concurrent region. The speedup at a parallel code granularity of 18 is 2.4. At a parallel code granularity of 9, the 50,000 cycles serialization threshold version issues a serializing recompilation. The speedup at a parallel code granularity of 9 is 1.4. The 10,000 cycles serialization threshold version issues a serializing recompilation at parallel code granularity of 2; the resulting slowdown is 12% for a parallel code granularity of 4, 22% for a parallel code granularity of 3, and 15% for a parallel code granularity of 2. The performance of the 10,000 cycles version drops below the performance of the sequential version, since (i) a certain amount of samples must be collected before the data is reported to the global performance database and (ii) a new value is only reported to the global database if the difference between the previous average execution time is above a certain threshold.

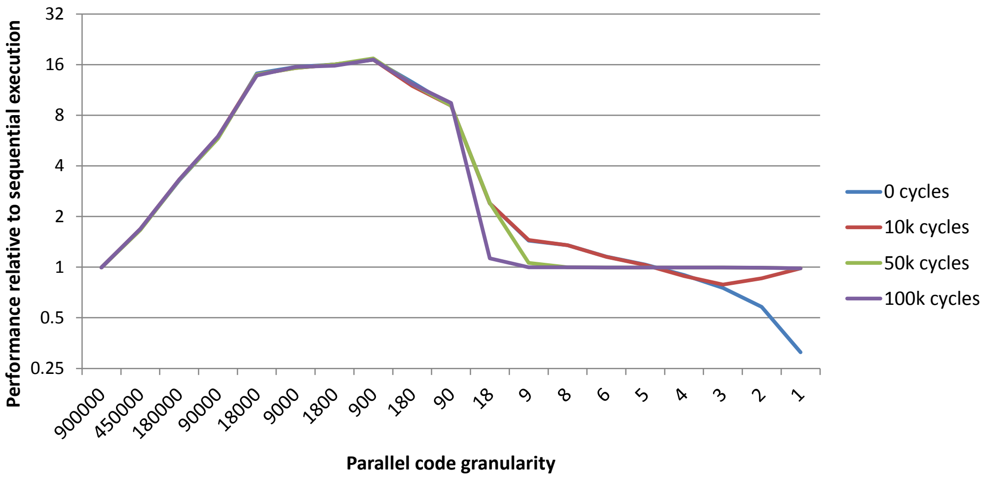


Figure 7.5: Determining the serialization threshold.

Figure 7.6 shows the relative performance of the parallel execution of the synthetic benchmark compared to the sequential execution for an increasing parallel code granularity.

Figure 7.6 illustrates the effects of the different serialization thresholds for an increasing parallel code granularity. I.e., all versions, except for the version which uses 0 cycles for the serialization threshold is initially compiled to sequential code. An increasing parallel code granularity results in a longer concurrent region execution time. If the concurrent region execution time exceeds the serialization threshold,

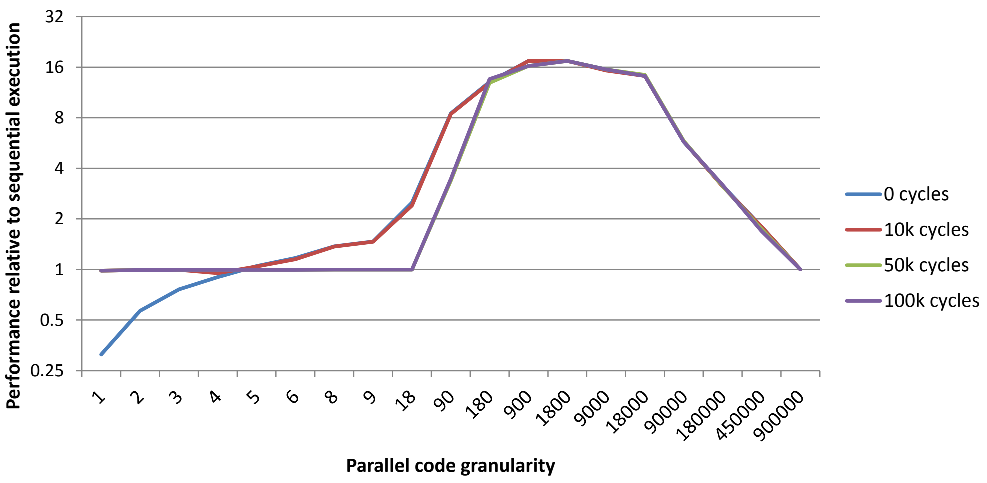


Figure 7.6: Determining the parallelization threshold.

the JIT compiler compiles the concurrent region to parallel code.

As shown in Figure 7.6 the 10,000 cycles serialization threshold version switches exactly to the parallel execution when the performance of the all-parallel version (0 cycles) reaches the performance of the sequential version. This happens at a parallel code granularity of 5. The 50,000 cycles version and the 100,000 cycles version issue a parallelizing recompilation at a parallel code granularity of 90.

To summarize, the serialization threshold determines if a concurrent region is executed sequentially or in parallel. The serialization threshold is reported to the user of the Java VM at the deployment time. The Java VM user can provide a multiplication factor (>1) that determines the efficiency at which parallel code is executed. For example, a multiplication factor of 10 results in a behavior as shown in Figure 7.5 and Figure 7.6.

7.5 Java Grande benchmark results

We use the Java Grande benchmarks to evaluate the performance of our system and compare the performance to the standard Java version of the Java Grande benchmarks.

7.5.1 Experimental methodology

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 statements are parallelized by annotating parallel loops with the `@Concurrent` annotation. To provide the opportunity to merge concurrent regions, we manually unroll the parallel loops. We have to do loop unrolling manually, since loop unrolling can produce unstable code in the Jikes RVM. We unroll all parallel loops of all benchmarks by a factor of eight. As a result, the PCEO algorithm can coarsen the given parallel code granularity by a factor of eight.

Each benchmark is executed with the same Jikes RVM configuration on both benchmark platforms. In particular, we use a profiling interval 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 also 0.99 for Y. Figure 7.7 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 a 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 Java VM is equipped with this merge function by default.

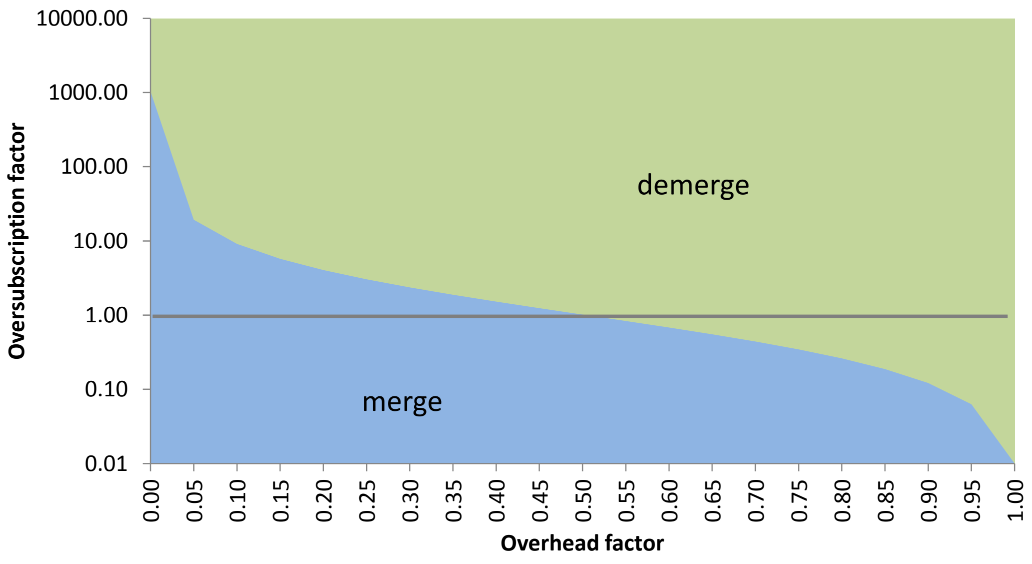


Figure 7.7: Used merge function.

All presented results are the average over 10 performance runs; we use 10 warmup runs, which are executed prior to the performance runs to measure steady-state performance [35]. The warmup runs and the performance runs are executed in the same Java VM instance.

The next section presents performance results for the two different hardware platforms that are described at the beginning of this chapter. We use two benchmark platforms to show that our approach provides performance portability, i.e., the same code performs well on different target platforms.

7.5.2 Results

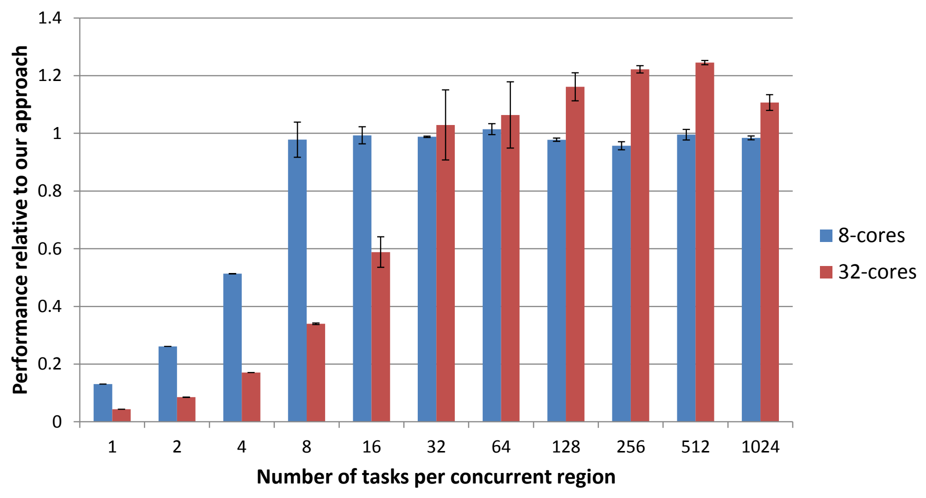
Figure 7.8 compares the performance of our approach with the performance of standard Java for the `crypt`, `lu`, `series`, `matmult`, `sor`, and `montecarlo` benchmark of the Java Grande benchmarks.

The x-axes in Figure 7.8(a) – Figure 7.8(f) show the number of tasks into which a parallel loop is split for the standard Java version. For example, a value of 2 on the x-axes means that the iteration space of the loop is assigned to two task objects. Since some parallel loops are inner loops, selecting a good number of tasks is non-trivial. The y-axes in Figure 7.8(a) – Figure 7.8(f) show the relative execution time to the version using concurrent `for` loops. More specifically, a value of 1 on the y-axes corresponds to the execution time of our approach. A value smaller than 1 means that the standard Java version is slower than our approach. A value larger than 1 means that the standard Java version is faster than our approach. The error bars indicate the standard deviation. Note that our version automatically determines the number of parallel tasks.

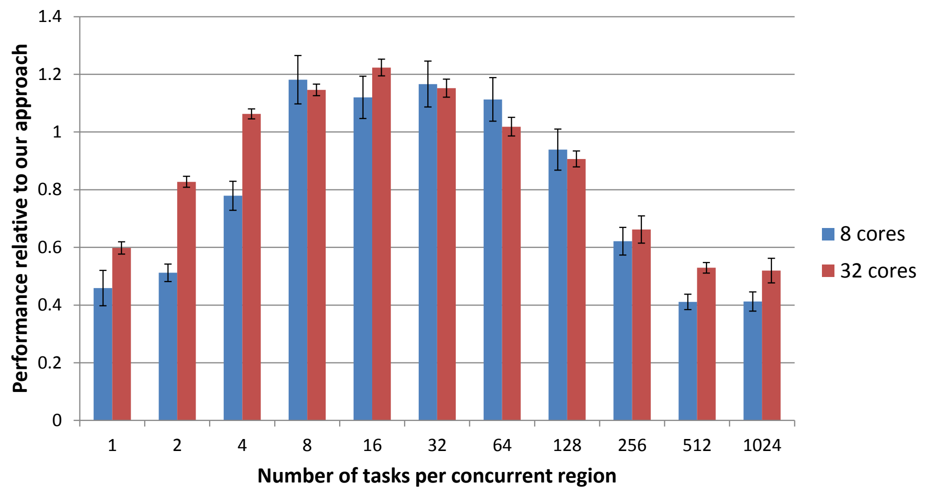
Note that the 8-core system can reach a relative performance of 1 potentially earlier than the 32-core system, since the 8-core system requires only 8 tasks to keep all execution units busy. The 32-core system requires at least 32 tasks. Table 7.3 shows the number of generated tasks for the standard Java version for the 10 warmup runs as well as the 10 performance runs. Table 7.3 also shows the number of generated tasks for our approach.

The `crypt` benchmark performs encryption and decryption using the International Data Encryption Algorithm. The algorithm is applied to an N byte array. The parallel versions distribute the work to threads in a block fashion. Our approach generates 219,000 tasks on the 8-core system. The resulting performance is similar to the standard Java version if more than 4 tasks are used to process the work. Our approach generates 1,305,000 tasks on the 32-core system. The parallel task granularity that results from the merging function is slightly too fine grained. As a result, the performance of our approach on the 32-core system is around 20% lower compared to the standard Java version. Note that the profiling overhead cannot introduce the 20% overhead, since the performance of the standard Java version using 32 tasks is similar to the performance of our approach.

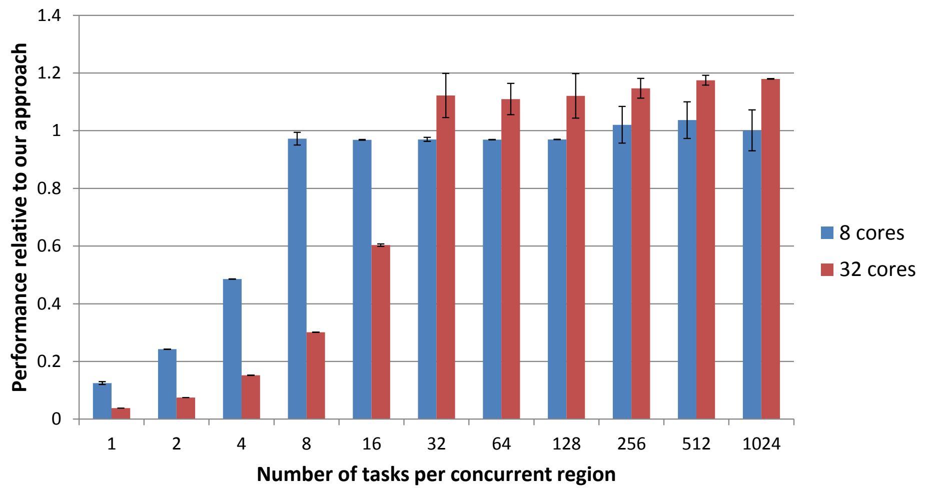
The `lu` benchmark solves a N $\times$ N linear system by LU Factorization followed



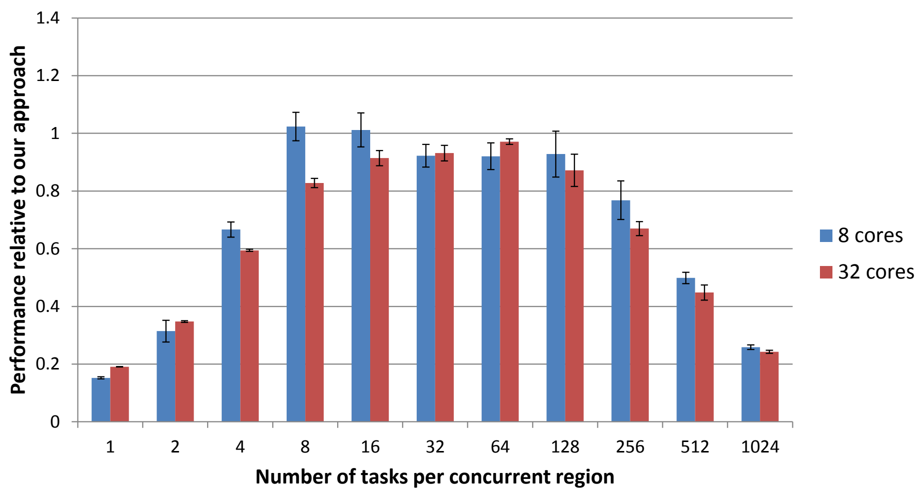
(a) `crypt` benchmark results.



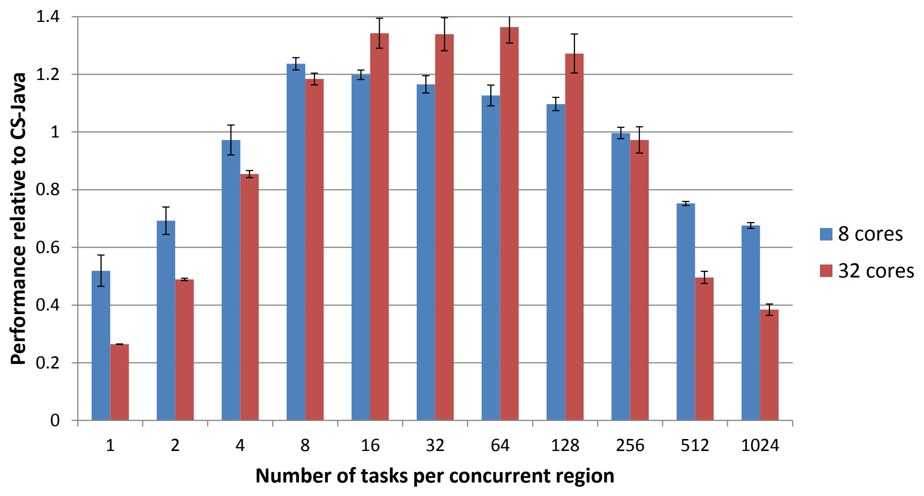
(b) `lu` benchmark results.



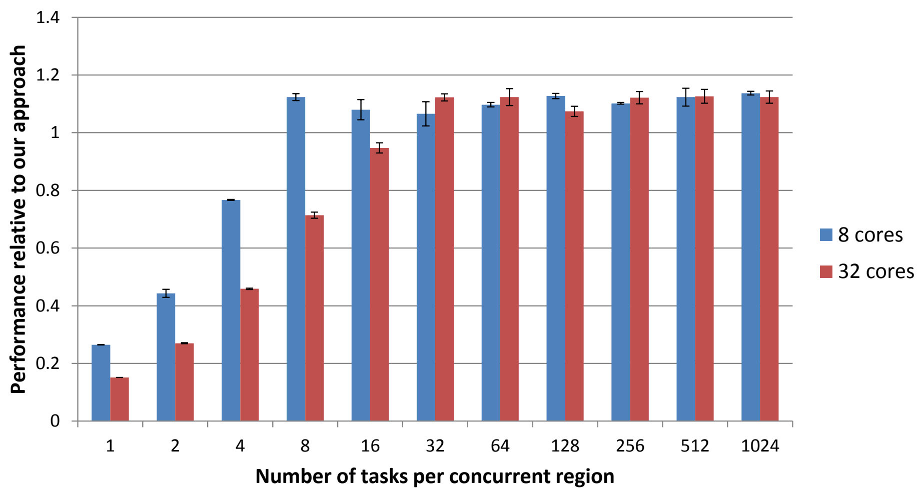
(c) `Series` benchmark results.



(d) `matmult` benchmark results.



(e) `sor` benchmark results.



(f) `montecarlo` benchmark results.

Figure 7.8: Performance results of the Java Grande benchmark Suite.

by a triangular rule. The only part of the `lu` benchmark that is parallelized is the factorization, which is implemented as a parallel loop that is contained in another loop. As a result, choosing a good number of tasks to which the iterations of the parallel loop are distributed is non-trivial. Our approach generates 1,500,000 tasks on the 8-core system and 1,300,000 tasks on the 32-core system. As shown in Figure 7.8(b) both numbers are slightly too high to achieve the maximum measured performance of the standard Java versions. The standard Java version on the 8-core system performs best when the parallel loop is split into 8 tasks. On the 32-core system, the standard Java version performs best when the parallel loop is split into 16 tasks. Using 128 tasks and more causes a significant performance loss on the 8-core system as well as on the 32-core system. Note that the `crypt` benchmark performed best using 512 tasks.

The `series` benchmark computes the first N coefficients of the function $f = (x+1)^x$ in an interval of 0.2. The `series` benchmark shows a similar performance characteristic as the `crypt` benchmark. Our approach creates 312,000 task objects on 32-core system as well as on the 8-core system. The number is large compared to the standard Java version, which generates at most 22,000 parallel tasks. Our approach merges all 8 concurrent regions into a single concurrent region. As a result, the number of parallel tasks cannot be reduced any further. Since the computations of the `series` benchmark are regular, i.e., the execution times of the tasks are similar, a small number of parallel tasks provides the best performance. However, our approach performs equally well as the standard Java version on the 8-core system. Our approach is 10% – 20% slower compared to the standard Java version on the 32-core system if 32 tasks or more are used to execute the parallel loop.

The `matmult` benchmark performs matrix multiplication of an unstructured sparse N $\times$ N matrix. The matrix is divided into blocks and each thread computes the results for a block. The benchmark performs the matrix multiplication 200 times. The `matmult` benchmark is the only benchmark in which our approach outperforms the standard Java version on the 32-core system for all evaluated task configurations. Our approach generates 185,000 tasks on the 32-core system. Our approach performs equally well as the standard Java version on the 8-core system if the work is divided into 8 or 16 tasks. For all other configurations, our approach outperforms standard Java. As shown in Figure 7.8(d), overspecifying parallelism results in a significant performance penalty.

The `sor` benchmark performs 100 iterations of the successive over-relaxation of an N $\times$ N grid. The parallel implementation distributes the work in a block fashion. Our approach is significantly slower (up to 40%) compared to the standard Java version on both systems. The reason is that our system merges all 8 concurrent regions on both systems. However, the number of generated tasks is significantly larger than in the standard Java version. However, recall that the merging capability of parallel loops is limited to the static loop unrolling by a factor of 8 in the source code. If loop unrolling could be performed in the Jikes RVM, our system can merge more concurrent regions. Fixing the loop unrolling compiler pass in the Jikes

RVM is subject to future work. Similar to the `matmult` benchmark, overspecifying parallelism significantly decreases the performance of the standard Java version on both platforms.

The `montecarlo` benchmark performs a financial simulation. The `montecarlo` benchmark shows a similar behavior as the `crypt` and the `series` benchmark. Our system generates 187,000 tasks on the 8-core as well as on the 32-core system. Similar to the `sor` benchmark, all concurrent regions are merged. However, the number of generated tasks is significantly larger than in the standard Java version. However, our system is only around 10% slower than the standard Java version.

Benchmark	Our approach 8	Our approach 32	1	2	4	8	16	32	64	128	256	512	1024
`crypt`	219	1,305	0.04	0.08	0.16	0.32	0.64	1.28	1.92	4.48	7.04	12.2	21.2
`lu`	1,509	1,296	39.9	69.8	129	247	468	805	1,443	2,845	5,637	9,990	9,990
`series`	312	312	0.04	0.06	0.14	0.30	0.62	1.26	1.90	3.18	8.30	13.4	22.4
`matmult`	65	185	4	8	16	32	64	128	256	512	1,024	2,048	4,096
`sor`	935	935	8	12	28	44	108	140	276	556	1,212	3,000	5,996
`montecarlo`	187	187	0.04	0.08	0.16	0.32	0.64	1.28	1.92	4.48	7.04	12.2	21.2

Table 7.3: Number of thousand generated parallel tasks for standard Java.

7.6 Dynamic recompilation overhead

This section compares the performance characteristics of the recompilation system of the Java version and our system. Figure 7.9 shows the total time that is used by the optimizing compiler. The data is obtained by running the application on the 32-core system using 32 threads with the `-X:vm:measureCompilationTime=true`. We use 32 threads on the 32-core system to measure the compilation time on a fully-loaded system. Since recompilation is done by a separate thread while the application is running, this setting results in the potentially largest compilation time. The numbers shown are the average over 5 runs. The error bars indicate the standard deviation.

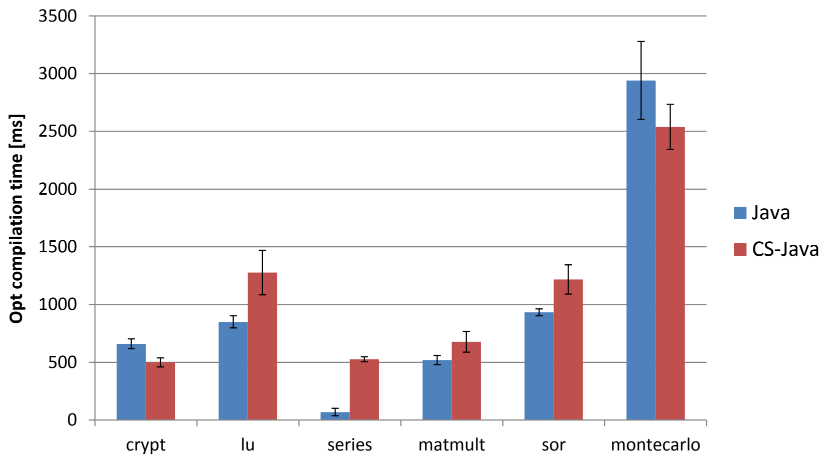


Figure 7.9: Total compilation time for the optimizing compiler.

As shown in Figure 7.9, the dynamic compilation time of the Java versions and our system are similar for the `crypt` and the `matmult` benchmark. Both, `crypt` and `matmult` perform approximately 9-10 recompilations due to concurrent region merging on the 8-core as well as on the 32-core system. The concurrent statement version of the `lu` benchmark incurs the largest compilation time overhead compared the standard Java version. The reason is that the number of iterations of the parallel loop (which is nested inside another loop) is different for each invocation. Consequently, the `lu` benchmark incurs a high number of recompilations (approx. 90) due to merging and demerging concurrent regions. The dynamic compilation time of the `series` benchmark executed with our system is significantly higher as for the standard Java version. The number of recompilations due to concurrent region optimizations is only 10. The reason is that the JIT compiler of our system compiles more methods, since the first iteration of the `series` benchmark is executed sequentially. The reason is that the Jikes RVM does not support on-stack replacement in the chosen configuration. The `sor` benchmark shows a similar characteristic as the `matmult` benchmark. The total number of recompilations due to concurrent region execution optimizations is 10. Finally, the `montecarlo` benchmark shows an equal compilation time behavior for the standard Java version and our approach. Note that the error bars overlap. As a result, the dynamic compilation times can be considered equal.

8

Concluding remarks

This thesis presents a novel approach for an *automatic*, *dynamic*, and *feedback-directed* optimization of parallel Java code. 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 enables the shifting of the setting of how parallel code is executed from the application/library level to the Java Virtual Machine (Java VM) level. The second principle is the concurrency-awareness of the runtime system, the Java VM. Concurrency-awareness enables the runtime system to gather profile information that is relevant to optimize the performance of parallel code.

The combination of both principles enables the *runtime system* to manage the execution of parallel code based on the collected profile information. In particular, the runtime system determines the assignment of parallel code to tasks. Since the Java VM manages the execution of parallel code, the approach is automated. Since the profile is collected at run-time, the approach is dynamic. Since the runtime system constantly profiles the parallel code setting and incorporates information about the last setting into the decision of a future, optimized setting, the approach is feedback-directed.

8.1 Summary and contributions

The thesis validates the thesis statement in Section 1.2 as follows:

1. The extension of the Java Programming Language by *concurrent statements* provides the separation of concern of declaring parallel code from its execution. A concurrent statement *may* be executed in parallel. The runtime system makes the final decision which thread executes a concurrent statement (Chapter 4).

2. The extension to the dynamic compilers, the adaptive optimization system, and profiling system system makes the runtime system concurrency-aware (Chapter 4, Chapter 5).

3. The parallel code execution optimization algorithm automatically optimizes parallel code towards a pre-defined metric (Chapter 6).
4. The performance evaluation shows that our system achieves similar or better performance compared to hand-optimized code. Furthermore, the performance evaluation shows the effectiveness of the presented parallel code execution optimization on two different hardware platforms (Chapter 7).

8.2 Expandability

There are various opportunities for future work. For example, the existing profiling infrastructure can be extended to measure additional metrics that can influence the execution time of parallel code. Examples of such metrics are the time spent in critical sections, the contention on locks, false sharing, architecture-specific performance bottlenecks such as the off-chip bandwidth, or the memory layout on non-uniform memory architectures. This information can either be gathered by extending the instrumenting profiler or by using hardware performance counters, which are available on most modern processors.

Another direction for future work is to compare the code quality of parallel code that is inlined into a concurrent region and can therefore profit from optimizations that can be performed across thread boundaries against parallel code that is not inlined.

A

Detailed code examples

```
void process(int);
 0: iload_1
 1: aload_0
 2: getfield #21 //Field threshold:I
 5: if_icmpge 15
 8: aload_0
 9: invokespecial #23 //Method executeSequential:()V
 12: goto 57
 15: iload_1
 16: iconst_2
 17: idiv
 18: istore_2
 19: new #25 //class Task
 22: dup
 23: iconst_0
 24: iload_2
 25: invokespecial #27 //Method Task."<init>":(II)V
 28: astore_3
 29: aload_0
 30: getfield #30 //Field threadPool:LThreadPool;
 33: aload_3
 34: invokevirtual #32 //Method ThreadPool.submit:(LTask;)V
 37: new #25 //class Task
 40: dup
 41: iload_2
 42: iconst_1
 43: iadd
 44: iload_1
 45: invokespecial #27 //Method Task."<init>":(II)V
 48: astore_3
 49: aload_0
 50: getfield #30 //Field threadPool:LThreadPool;
 53: aload_3
 54: invokevirtual #32 //Method ThreadPool.submit:(LTask;)V
 57: return
```

Figure A.1: Full Bytecode representation of Figure 2.3(a).

```
void process(int);
  0: iload_1
  1: aload_0
  2: getfield #19 //Field threshold:I
  5: if_icmpge 15
  8: aload_0
  9: invokespecial #21 //Method executeSequential:()V
  12: goto 73
  15: iload_1
  16: iconst_2
  17: idiv
  18: istore_2
  19: new #23 //class Task
  22: dup
  23: iconst_0
  24: iload_2
  25: invokespecial #25 //Method Task."<init>":(II)V
  28: astore_3
  29: new #28 //class java/lang/Thread
  32: dup
  33: aload_3
  34: invokespecial #30 //Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
  37: astore 4
  39: new #23 //class Task
  42: dup
  43: iload_2
  44: iconst_1
  45: iadd
  46: iload_1
  47: invokespecial #25 //Method Task."<init>":(II)V
  50: astore 5
  52: new #28 //class java/lang/Thread
  55: dup
  56: aload 5
  58: invokespecial #30 //Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
  61: astore 6
  63: aload 4
  65: invokevirtual #33 //Method java/lang/Thread.start:()V
  68: aload 6
  70: invokevirtual #33 //Method java/lang/Thread.start:()V
  73: return
}
```

Figure A.2: Full Bytecode representation of Figure 2.4(a).

Bibliography

[1] Clang. http://clang.llvm.org/.

[2] .NET. http://www.microsoft.com/net.

[3] Message Passing Interface, 2008. http://www.mpi-forum.org/docs/docs.html.

[4] OpenMP Application Program Interface, 2012. http://openmp.org/wp/.

[5] Acar, U. A., Charguéraud, A., and Rainey, M. Oracle scheduling: controlling granularity in implicitly parallel languages. In *OOPSLA '11*.

[6] Adve, S. V. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[7] Adve, S. V., and Aggarwal, J. K. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*.

[8] Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R. K., and Yelick, K. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07*.

[9] Agarwal, S., Barik, R., Sarkar, V., and Shyamasundar, R. K. May-happen-in-parallel analysis of x10 programs. In *PPoPP '07*.

[10] Agrawal, K., Leiserson, C. E., He, Y., and Hsu, W. J. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst. 26*, 3.

[11] Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The Jalape no virtual machine. *IBM Syst. J. 39*, 1 (2000).

[12] Angerer, C., and Gross, T. Exploiting task order information for optimizing sequentially consistent java programs. In *PACT '11*, ACM.

[13] Angerer, C. M., and Gross, T. R. Exploiting task order information for optimizing sequentially consistent java programs. *PACT'11*.

[14] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. Adaptive optimization in the jalapeno JVM.

[15] ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*.

[16] BARIK, R., AND SARKAR, V. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT '09*, ACM.

[17] BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04*.

[18] BLACKBURN, S. M., AND MCKINLEY, K. S. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08*.

[19] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5.

[20] BLUMOFE, R. D., AND LISIECKI, P. A. Adaptive and reliable parallel computing on networks of workstations. In *ATEC '97*.

[21] BLUMOFE, R. D., AND PAPADOPOULOS, D. The performance of work stealing in multiprogrammed environments (extended abstract). In *SIGMETRICS '98/PERFORMANCE '98*.

[22] BLUNDELL, C., MARTIN, M. M., AND WENISCH, T. F. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA '09*.

[23] BOEHM, H.-J. Threads cannot be implemented as a library. In *PLDI '05*.

[24] BURTON, F. W., AND SLEEP, M. R. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*.

[25] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40*, 10 (2005).

[26] CHEN, S., GIBBONS, P. B., KOZUCH, M., LIASKOVITIS, V., AILAMAKI, A., BLELLOCH, G. E., FALSAFI, B., FIX, L., HARDAVELLAS, N., MOWRY, T. C., AND WILKERSON, C. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07*.

[27] CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst. 25* (November 2003).

[28] CHOW, J.-H., AND HARRISON, III, W. L. Compile-time analysis of parallel programs that share memory. In *POPL '92*.

[29] DING, X., WANG, K., GIBBONS, P. B., AND ZHANG, X. Bws: balanced work stealing for time-sharing multicores. In *EuroSys '12*.

[30] Dwyer, M. B., and Clarke, L. A. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94.

[31] Feeley, M. A message passing implementation of lazy task creation. In *In Parallel Symbolic Computing: Languages, Systems, and Applications (US/-Japan Workshop Proceedings). Springer-Verlag Lecture Notes in Computer Science 748* (1993).

[32] Fink, S. J., Knobe, K., and Sarkar, V. Unified analysis of array and object references in strongly typed languages. In *SAS '00*, Springer-Verlag.

[33] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5 (1998).

[34] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[35] Georges, A., Buytaert, D., and Eeckhout, L. Statistically rigorous java performance evaluation. In *OOPSLA '07.*

[36] Gharachorloo, K., Gupta, A., and Hennessy, J. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS-IV.*

[37] Gniady, C., and Falsafi, B. Speculative sequential consistency with little custom storage. In *PACT '02.*

[38] Gniady, C., Falsafi, B., and Vijaykumar, T. N. Is sc + ilp = rc? In *ISCA '99.*

[39] Goldstein, S. C., Schauser, K. E., and Culler, D. E. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing 37* (1996), 5–20.

[40] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java(TM) Language Specification, The 3rd Edition.* Addison-Wesley Professional, 2005.

[41] Grunwald, D., and Srinivasan, H. Data flow equations for explicitly parallel programs. *SIGPLAN Not. 28* (July 1993).

[42] Hall, M. W., and Martonosi, M. Adaptive parallelism in compiler-parallelized code. In *IN PROC. OF THE 2ND SUIF COMPILER WORKSHOP.*

[43] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T. Backtracking-based load balancing. In *PPoPP '09.*

[44] Hoeflinger, J. P., and De Supinski, B. R. The openmp memory model. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming* (2008), IWOMP'05/IWOMP'06.

[45] HUELSBERGEN, L., LARUS, J. R., AND AIKEN, A. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings of the 1994 ACM conference on LISP and functional programming.*

[46] JOISHA, P. G., SCHREIBER, R. S., BANERJEE, P., BOEHM, H.-J., AND CHAKRABARTI, D. R. On a technique for transparently empowering classical compiler optimizations on multithreaded code. *ACM Trans. Program. Lang. Syst. 34*, 2.

[47] JOISHA, P. G., SCHREIBER, R. S., BANERJEE, P., BOEHM, H. J., AND CHAKRABARTI, D. R. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL '11*, ACM.

[48] KAMINSKY, A. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *IPDPS '07.*

[49] KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. Effective null pointer check elimination utilizing hardware trap. In *ASPLOS-IX*, ACM.

[50] KNOOP, J., STEFFEN, B., AND VOLLMER, J. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst. 18* (May 1996).

[51] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the Java HotSpot^{TM}client compiler for java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008).

[52] KRISHNAMURTHY, A., AND YELICK, K. Optimizing parallel spmd programs. In *LCPC'94* (1994).

[53] KUMAR, K. V. S. When and what to compile/optimize in a virtual machine? *SIGPLAN Not.*.

[54] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[55] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (1979).

[56] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04.*

[57] LAU, J., ARNOLD, M., HIND, M., AND CALDER, B. Online performance auditing: using hot optimizations without getting burned. In *PLDI '06.*

[58] LEA, D. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, ACM.

[59] LEE, J., MIDKIFF, S. P., AND PADUA, D. A. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC '97*, Springer-Verlag.

[60] LEE, J., PADUA, D. A., AND MIDKIFF, S. P. Basic compiler algorithms for parallel programs. In *PPoPP '99*, ACM.

[61] Leijen, D., Schulte, W., and Burckhardt, S. The design of a task parallel library. *SIGPLAN Not. 44*, 10 (2009).

[62] Long, D., and Clarke, L. A. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the symposium on Testing, analysis, and verification* (1991), TAV4.

[63] Lopez, P., Hermenegildo, M., and Debray, S. A methodology for granularity-based control of parallelism in logic programs. *J. Symb. Comput. 21*, 4-6 (June 1996).

[64] Manson, J., Pugh, W., and Adve, S. V. The Java memory model. In *POPL '05*, ACM.

[65] Mao, F., Zhang, E. Z., and Shen, X. Influence of program inputs on the selection of garbage collectors. In *VEE '09.*

[66] Midkiff, S., and Padua, D. Issues in the optimization of parallel programs. In *International Conference on Parallel Processing* (1990).

[67] Midkiff, S. P. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques.* Morgan & Claypool Publishers, 2012.

[68] Mohr, E., Kranz, D. A., and Halstead, Jr., R. H. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming.*

[69] Muchnick, S. S. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., 1997.

[70] Namjoshi, M. A., and Kulkarni, P. A. Novel online profiling for virtual machines. In *VEE '10.*

[71] Nichols, B., Buttlar, D., and Farrell, J. P. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[72] Pai, V. S., Ranganathan, P., Adve, S. V., and Harton, T. An evaluation of memory consistency models for shared-memory systems with ilp processors. In *ASPLOS-VII.*

[73] Pehoushek, J. D., and Weening, J. S. Low-cost process creation and dynamic partitioning in qlisp. In *Proceedings of the US/Japan workshop on Parallel Lisp on Parallel Lisp: languages and systems* (1990).

[74] Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., and Holmes, D. *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[75] Raman, A., Kim, H., Oh, T., Lee, J. W., and August, D. I. Parallelism orchestration using dope: the degree of parallelism executive. In *PLDI '11.*

[76] Raman, A., Zaks, A., Lee, J. W., and August, D. I. Parcae: a system for flexible parallel execution. In *PLDI '12.*

[77] RUGINA, R., AND RINARD, M. C. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst. 25* (January 2003).

[78] SANCHEZ, D., YOO, R. M., AND KOZYRAKIS, C. Flexible architectural support for fine-grain scheduling. In *ASPLOS '10*.

[79] SARKAR, V. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC '97*, Springer-Verlag.

[80] SATOH, S., KUSANO, K., AND SATO, M. Compiler optimization techniques for openmp programs. *Sci. Program. 9* (August 2001).

[81] SCHNEIDER, F. T., PAYER, M., AND GROSS, T. R. Online optimizations driven by hardware performance monitoring. In *PLDI '07*.

[82] SHASHA, D., AND SNIR, M. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2.

[83] SHIRAKO, J., ZHAO, J. M., NANDIVADA, V. K., AND SARKAR, V. N. Chunking parallel loops in the presence of synchronization. In *ICS '09*, ACM.

[84] SMITH, L. A., BULL, J. M., AND OBDRZÁLEK, J. A parallel Java grande benchmark suite. In *Supercomputing '01*.

[85] SRINIVASAN, H., HOOK, J., AND WOLFE, M. Static single assignment for explicitly parallel programs. In *POPL '93*, ACM.

[86] SRINIVASAN, H., AND WOLFE, M. Analyzing programs with explicit parallelism. In *LCPC'92*.

[87] SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst. 27*, 4.

[88] SUGANUMA, T., YASUE, T., AND NAKATANI, T. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*.

[89] SULEMAN, M. A., QURESHI, M. K., KHUBAIB, AND PATT, Y. N. Feedback-directed pipeline parallelism. In *PACT '10*.

[90] SULEMAN, M. A., QURESHI, M. K., AND PATT, Y. N. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *ASPLOS XIII*.

[91] SURA, Z., FANG, X., WONG, C.-L., MIDKIFF, S. P., LEE, J., AND PADUA, D. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP '05*, ACM.

[92] TARDIEU, O., WANG, H., AND LIN, H. A work-stealing scheduler for x10's task parallelism with suspension. In *PPoPP '12*.

[93] TIAN, X., GIRKAR, M., SHAH, S., ARMSTRONG, D., SU, E., AND PETERSEN, P. Compiler and runtime support for running OpenMP programs on pentium-and itanium-architectures. In *HIPS '03*, IEEE.

[94] von Praun, C., Cain, H. W., Choi, J.-D., and Ryu, K. D. Conditional memory ordering. In *ISCA '06*.

[95] Wang, L., Cui, H., Duan, Y., Lu, F., Feng, X., and Yew, P.-C. An adaptive task creation strategy for work-stealing scheduling. In *CGO '10*.

[96] Weening, J. S. *Parallel execution of LISP programs*. PhD thesis, Stanford, CA, USA, 1989. UMI Order No: GAX89-25973.

[97] Whaley, J. A portable sampling-based profiler for java virtual machines. In *JAVA '00*.

[98] Yang, X., Blackburn, S. M., Frampton, D., Sartor, J. B., and McKinley, K. S. Why nothing matters: the impact of zeroing. In *OOPSLA '11*.

[99] Zhang, Y., Duesterwald, E., and Gao, G. R. Concurrency analysis for shared memory programs with textually unaligned barriers. In *LCPC'08*.

[100] Zhao, J., Shirako, J., Nandivada, V. K., and Sarkar, V. Reducing task creation and termination overhead in explicitly parallel programs. In *PACT '10*, ACM.

Curriculum Vitae

Albert Noll

April 20, 1981	Born in Tamsweg, Salzburg, Austria
1988 – 1991	Primary School in Sauerfeld, Austria
1991 – 1999	High school (Gymnasium) in Tamsweg, Austria
2000 – 2005	Bachelor in Telematics
2005 – 2007	Dipl. Ing. in Telematics, University of Technology Graz, Austria
2007 – 2013	Doctoral Studies at ETH Zurich, Switzerland