Refactoring Traces to Identify Concurrency Improvements

Indigo Orton
indigo.orton@cl.cam.ac.uk
University of Cambridge
Department of Computer Science and Technology
Cambridge, UK

Alan Mycroft
alan.mycroft@cl.cam.ac.uk
University of Cambridge
Department of Computer Science and Technology
Cambridge, UK

ABSTRACT

It is often difficult to analyse why a program executes more slowly than intended. This is particularly true for concurrent programs. We describe and evaluate a system, *Rehype*, which takes Java programs, performs low-overhead tracing of method calls, analyses the resulting *trace-logs* to detect inefficient uses of concurrency constructs, and suggests source-code-oriented improvements. *Rehype* deals with task-based concurrency, specifically a future-based model of tasks. Implementing the suggested improvements on an industrial API server more than doubled request-processing throughput.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; Object oriented languages; Concurrent programming languages; **Software performance**; **Dynamic analysis**; • **Theory of computation** → *Concurrency*; **Program analysis**.

KEYWORDS

Concurrent improvements, Performance prediction, Trace refactoring, Task-based concurrency

ACM Reference Format:
Indigo Orton and Alan Mycroft. 2021. Refactoring Traces to Identify Concurrency Improvements. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21), July 13, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3464971.3468420

1 INTRODUCTION

We address inefficient use of task-based concurrency within existing Java programs that use thread pools to schedule task execution. Task-based concurrency (Section 2) simplifies parallelising programs as it is conceptually and logistically straightforward. It is widely used in industrial software. As of writing (May 2021), public GitHub [4] repositories had 938 196 instances of `.java` files importing `java.util.concurrent.ExecutorService` (a core Java 8 utility for *future*-based task concurrency, it is the base interface for spawning tasks). Furthermore, despite the increasing popularity of reactive programming, task-based concurrency remains a prevalent approach to concurrency within Java 8 (and beyond). Moreover, over the past decade numerous languages (including versions of Rust, C#, Dart, Python, Hack, JavaScript, Scala, and C++) have added support for the `async`/`await` concurrency pattern – in some cases this pattern is the core concurrency approach for the language. The `async`/`await` pattern is task-based concurrency, indeed in many languages and runtimes it is a syntactically sugared version of futures.

Inefficient task patterns are easy to fall into and regularly occur in industrial software. A common inefficient pattern is wait-limited tasks. Wait-limited tasks do little computation and instead spend the majority of their execution waiting on other tasks to finish. This is inefficient because a waiting task consumes system resources (e.g. a Java thread) without progressing the program's computation. In a server handling multiple requests, this performance loss manifests as a reduction in *throughput*; it often has no visible effect on either wall-clock or CPU time taken for a solitary request. The cause is clogging of system-wide thread pools. For example, consider a thread pool of eight worker threads with four physical CPU cores; if seven threads are waiting for another task to complete, while only one thread is doing work, then only one core is being used, thus achieving a quarter of the potential performance. The seven waiting threads remain idle, consuming system resources (memory), even while other tasks may be queued in the thread pool, waiting to be executed. In this scenario, refactoring wait-limited tasks can improve performance (for metrics of throughput and wall-clock time, but not classically-accounted CPU time) by freeing threads for other tasks. While these inefficiencies may be resolved by using a different concurrency model (e.g. reactive programming), such refactoring is impractical for large programs, and may introduce other issues. Incremental refactoring reduces risk, while prioritising refactorings improves return on investment – performance benefit per unit developer time.

In many cases there are relatively low-cost solutions (i.e. minimal, localised, source-code improvements) to these inefficient patterns. For example, wait-limited tasks can be either inlined or replaced with combinators (e.g. `CompletableFuture` in Java 8). The two primary inhibitors to improving concurrency use are the difficulties in identifying inefficient tasks and prioritising improvements. Identifying inefficient concurrency patterns statically is difficult as, by their nature, concurrency and performance patterns emerge at runtime. Prioritising improvements is important in industrial contexts where there are limited developer resources and substantial amounts of source code.

We introduce and evaluate *Rehype*, a system[1] that traces program execution and performs automatic analysis on runtime execution traces, identifies instances of inefficient concurrency patterns, and suggests potential localised source-code improvements. It estimates

[1]For reference, *Rehype* is implemented in approx. 30k lines of Rust code.

the effect of each suggested improvement on the program's performance as a set of metrics, *without re-executing the program*. A developer can then sort and select suggested improvements, based on the metrics, to optimise for their desired performance attributes. *Rehype* produces *improvement specifications* which define how to implement an improvement; it does not generate source code patches. A companion paper [9], also appearing at FTfJP'21, describes a system, *Scopda*, for generating source code patches from *Rehype* improvement specifications.

Rehype consists of two main components: a low-overhead *tracer* and an *analyser* (which is the primary focus of this paper). The *tracer* (Section 3) automatically instruments Java programs, based on a *trace-config*, and records specific execution events (such as timestamped entry and exit of methods) that are then stored in a *trace-log*. The *analyser* (Section 4) takes a *trace-log*, identifies potential improvements (by moving sets of thread-contiguous events within the *trace-log*) and estimates their performance effects by recalculating timestamps.

While *Rehype* operates solely on dynamic data, in the form of *trace-logs*, we give a static code example of an improvement (Fig. 2) to provide intuition (noting that the absence of branching and looping in the example hides various issues). Importantly, *Rehype* aims to suggest only *statically implementable* improvements. If multiple tasks are spawned with identical dynamic context (functions active on the stack) – perhaps due to a source-code loop construct – then they must be treated consistently by the *analyser* (e.g. they must all be inlined, or none of them). We accordingly distinguish the idea of *optimisation*, such as *Rehype*'s *convert-to-inline*, from its instances (here called *improvements*) which are concrete modifications of *trace-logs*. For the case above, the improvement might be expressed as "inline every task spawned at dynamic context X".

A key idea is that *Rehype* identifies improvements by estimating the performance effects of implementing them. This estimation is performed by manipulating a *trace-log*. It occurs offline and does not require re-executing the program. Using an estimation-based approach enables *Rehype* to identify improvements in programs of varying complexity and for a variety of patterns, as it does not *detect* specific patterns, but *tests* specific improvements. This approach is well suited to concurrency performance optimisation as it can test the effect of implementing multiple interfering improvements. The *analyser* can accurately estimate the effects of changes to concurrency as the *trace-log* contains task life-cycle events that define the inter-thread concurrent dependencies, allowing the *analyser* to account for the concurrent behaviour.

Implementing a combination of improvements suggested by *Rehype* for a large server program more than doubled its request processing throughput (Section 5). While there is a lot of hype around increasing concurrency, *Rehype*'s results show that *reducing* concurrency can significantly improve performance in resource-constrained programs, such as application API servers.

Finally, Section 6 positions this work among related work, Section 7 discusses some questions regarding concurrency constructs, and Section 8 concludes the paper.

2 TARGET CONCURRENCY MODEL

Rehype is designed to analyse programs that use task-based concurrency with thread-pool scheduling. *Tasks* are concurrent constructs that enable scheduling the execution of a function (here called the *task-body function*) with some parameters. *Rehype* specifically analyses a *future*-based model of tasks, whereby spawning a task generates a *future* representing the return value of the task. When the task's result is retrieved from the future, the call blocks until the task has finished (providing synchronisation). *Rehype* assumes[2] that concurrent communication occurs solely at task spawn and on retrieving future-based results[3]. Thus, in essence, tasks present a pure-functional interface with the exception of operations that do not affect the result of the program, such as logging.

Intuitively, the mental model is of tasks corresponding to pure functions that are called and eventually return a single result (via a future). Specifically a task's logical behaviour (modulo memory locations, etc) is determined by its parameters. In practice, tasks will rarely be true pure functions, they may depend on, and mutate, some external state (such as a database). However, the key is that such state dependence is not relied upon for inter-task communication (i.e. a task does not rely on another, concurrently executing, task having edited that state).

Thread pools contain a fixed number of worker *threads* and a queue of *tasks* to execute. We assume that all *threads* in a *thread pool* are solely used for executing *tasks*. In this paper *threads* refer to the source-code-level logical threads. Although threads may be executed on various physical CPU cores, depending on the operating system scheduler, this is below the level of detail considered in this paper.

Threads exist in one of three states: 1) *active* when executing a task; 2) *waiting* when asleep waiting for the result of another task; and 3) *unoccupied* when waiting to be assigned a task by the thread pool. To maximise performance, programs should have most threads *active* most of the time. While *waiting*, a thread is consuming system resources but not actively progressing its current *task*. An *unoccupied* thread is available for use and consumes fewer system resources.

Java 8, the target runtime of this paper, has a standard task concurrency framework in the `java.util.concurrent` package [5]. The key primitives are `Future` and `ExecutorService`. `Future` is a handle for the result of a task, calling `Future.get()` sleeps the calling thread until the task is complete. The `ExecutorService` is used to schedule a task for execution and returns a `Future` for the task.

Java 8 includes the `Future` *interface* with various *implementations*. The basic implementation `FutureTask` handles executing a task and storing its result. A richer implementation `CompletableFuture` also implements combinator pattern utilities.

[2]Unfortunately, it is not possible to determine automatically whether a Java program adheres strictly to this concurrency model. While it is possible to check instances of the model being broken (e.g. clear usage of shared memory for coordination), proving the inverse (that the model is adhered to) is undecidable due to the aliasing problem (objects may be aliased and used between multiple tasks in non-obvious ways). However, new type systems, such as Rust's ownership system, may go some of the way towards enabling better model adherence checking.

[3]As an indicator, of the 938 196 Java files that import `ExecutorService` on GitHub (see introduction), 216 714 also contain the `synchronized` keyword, suggesting, though not confirming, that they do not adhere to this pure task model.

```
struct Event {
 // The type of event (e.g. FnStart, FnEnd, etc).
 enum EventType type;
 // The nanosecond time when the event occurred.
 uint64 time_stamp;
 // The id of the thread that this event occurred on.
 uint32 thread_id;
 union {
   // The id of the method the function event occurred in.
   uint32 function_event_method_id;
   // The id of the task the task event relates to.
   uint32 task_id;
 } aux_data;
}
```

Figure 1: A C-style definition of the data for each event.

```
class AgeQuerier {
  static ~~Future<~~DatabaseRecord~~>~~ queryDatabase(
      ExecutorService executorService, String id) {
    ~~return executorService.submit(() -> {~~
      DatabaseRecord queryResult = /* ... */;
      return queryResult;
    ~~});~~
  }

  static int queryAge(String id) {
    DatabaseRecord result = queryDatabase(id)~~.get()~~;
    return result.getAge();
  }
}
```

Figure 2: Example of the convert-to-inline optimisation.

Combinators allow wait-free programming; instead of a task waiting for a concurrently computed result, it merely terminates and a new task is spawned to use the result when it becomes ready. Combinators are used in one of the optimisations described in this paper. See Urma et al. [13] for more information on combinators in Java 8.

3 TRACER

The *tracer* generates a *trace-log* containing events that capture program-execution behaviour. It operates by inserting tracing code into a program in the style of aspect-oriented programming. There are three types of events: function, *task* life-cycle, and *thread-pool* scheduling events.

The tracer consists of an *instrumenter* and a *trace-logger*. The *instrumenter* takes a *trace-config* and inserts tracing code into a program’s `.jar` file. When the program is executed, the tracing code invokes *trace-logger* functions which save the events in a *trace-log*. The *trace-logger* is a runtime in-process utility that uses a client-server architecture. There is one client per thread and a single server to write to the *trace-log* on disk. Clients each maintain two buffers, when one is filled it is emptied into the *trace-log* by the server. This ensures no slow or blocking I/O operations occur on the instrumented program’s working threads. This process uses a bespoke *lockless buffer-swapping protocol* to avoid waits on the working threads.

The tracing code does not interfere with the normal output of the program. However, tracing does introduce overhead as it shares system resources; too much overhead can distort program behaviour (especially concurrency and performance behaviour). Such distortions can influence the efficacy of subsequent program analysis, hence the need for a low-overhead tracer. Experimental results suggest the overhead is in the order of tens of nanoseconds per event.

Events. All events contain a type byte, a nanosecond time-stamp of their occurrence, and the thread-id (unique identifier of a Java `Thread`) of the thread they occurred on (Fig. 1). Function events also contain a unique identifier of the function they occurred in and task events contain a unique identifier of the task they relate to (this is used to track concurrent execution across threads).

We assume a form of sequential consistency: that events executed by a single thread appear in event order, and with strictly increasing time-stamps, in the *trace-log*; events from separate threads may interleave in the *trace-log*, not necessarily in time-stamp order. Furthermore, events of different types may interleave in the *trace-log*.

4 ANALYSER

The analyser takes a *trace-log* and generates a list of suggested improvements along with their estimated effect on performance. The analyser individually estimates the effect of each possible *optimisation* for every task and then estimates the combined effect of sets of compatible *improvements*.

4.1 Optimisations

The analyser considers three optimisations: *convert-to-inline*, *hoist-branch*, and *convert-to-combinator*. The optimisations convert some concurrently executed tasks in a program to execute immediately at spawn (i.e. inline). Selectively removing concurrency can reduce the number of threads *waiting*, improving performance by freeing them to be used by other tasks. The optimisations differ on how the source code would be changed and thus which tasks are affected. Importantly, these optimisations do not affect the program’s functionality, provided the program adheres to the “all concurrent communication occurs through tasks (i.e. futures)” model explained in Section 2.

Convert-to-inline. The *convert-to-inline* optimisation involves executing a *task* immediately when spawned (on its spawning thread, instead of scheduling it to be executed on a separate thread). For example, in Fig. 2, pre-change `queryAge` invokes `queryDatabase` and immediately waits on the returned `Future`. Thus, for the task’s duration two threads are occupied, one purely for waiting. The *convert-to-inline* optimisation improves performance by reducing the number of occupied threads without affecting execution duration (or even reducing it depending on the task scheduling overhead). After the change, the top-level *task-body function*, `queryDatabase`, becomes synchronous.

A subtlety is that multiple tasks may be spawned at the same code location and would thus all be affected by the same code change. To account for this, the *convert-to-inline* optimisation is estimated for the set of tasks spawned at the same dynamic context.

Hoist-branch. The *convert-to-inline* optimisation can be somewhat of a blunt instrument given its all-or-nothing approach. The *hoist-branch* optimisation extends the *convert-to-inline* optimisation by identifying a subset of the tasks that can be inlined without

```
static Future<DatabaseRecord> parallelSum(ExecutorService es) {
  ~~return es.submit(() -> {~~
    ~~Future<Integer> first = es.submit(Example::longOperation1);~~
    ~~Future<Integer> second = es.submit(Example::longOperation2);~~
    ~~return first.get() + second.get();~~
  ~~});~~
  CompletableFuture<Integer> first = CompletableFuture.supplyAsync(
        Example::longOperation1, es);
  CompletableFuture<Integer> second = CompletableFuture.supplyAsync(
        Example::longOperation2, es);
  return second.thenCombine(first, (f, s) -> f + s);
}
```

Figure 3: Example of the convert-to-combinator optimisation.

affecting the other tasks. For example, when one subset of tasks performs extensive work, suitable for a task, and another subset returns quickly, it can be beneficial to inline only the latter subset. In the simple form, this difference in execution may be caused by an `if-then-else` (or `switch`) statement. The optimisation can be practically implemented by *hoisting* (moving) the `if-then-else` statement out of the task execution and to the task spawn code. Such cases of divergent executions can be identified in program traces based on the execution paths of the tasks (whereas *convert-to-inline* is purely based on the task-spawn dynamic context); when there is a hoist-able branching statement, the execution paths of the subsets of tasks will differ. The dynamic estimation of inlining a subset of tasks is the same as *convert-to-inline*, hence the optimisation is an extension of *convert-to-inline*. In fact, from a purely dynamic perspective, the optimisation could be called *convert-subset-to-inline*, however, the term *hoist-branch* better matches the practical change a developer would implement.

While there are more complex causes of branching, such as dynamic dispatch, the difference between task execution paths is similar whether due to a simple `if-then-else` or a more complex branching mechanism. As such, the *analyser* treats them similarly, though the static implementation may be more complex.

Convert-to-combinator. In Java, concurrency combinators can encode task scheduling and coordination (e.g. passing the result of one task as an argument to another task). Perfect use of combinators can result in sleep-free task concurrency as no thread ever waits for a task to finish. Instead of some code, x, waiting on a thread to use a task's result, a combinator represents x as a *task-body function* to be executed when the task result is ready.

Combinators are not as intuitive or straightforward to use as they require designing the task scheduling explicitly, instead of programming sequentially and relying on futures to determine synchronisation (whereby scheduling is handled by imperative code and threads waiting). Moreover, combinators are not always easy to practically retrofit to existing code.

The *convert-to-combinator* optimisation replaces *coordinator tasks*, tasks that primarily exist to coordinate other tasks, with combinators. This frees a thread to be used by other tasks. The optimisation determines the scheduling of the tasks spawned by the *coordinator task* and encodes it in a combinator (specifically a `CompletableFuture` in Java 8). Fig. 3 illustrates the optimisation in a simple parallel summation example.

A subtlety is that *Rehype* does not *detect* coordinator tasks, rather it estimates the effect of converting each applicable task (those that create and wait for another "sub-task") to a combinator. This embodies the "identify improvements by estimating the performance effects of implementing them" key idea described in the introduction; we leave determining and proposing concrete patches to the companion tool *Scopda* [9].

4.2 Estimation

To estimate the performance effects of an improvement δ (later we allow δ to be a set of improvements), on a program p, the analyser re-orders (and re-calculates the timings of) events in the *trace-log* to model executing p_δ, the idealised source-code program with δ implemented. That is, writing the *trace-log* of the program as $[\![p]\!]$, then the re-ordered *trace-log* $\beta_\delta\,[\![p]\!]$ approximates $[\![p_\delta]\!]$, where β_δ is a function that re-orders the *trace-log* for δ.

We implement β_δ using a *trace-DAG* of the *trace-log*. A *trace-DAG* is a structured representation of a *trace-log* which simplifies re-ordering as all time-stamps and thread-ids (see Fig. 1) are defined by the edges of the graph, making them relative values instead of absolute. The high-level β_δ process is: transform *trace-log* into a *trace-DAG*, apply δ changes to the *trace-DAG*, and transform the *trace-DAG* back into a *trace-log*. To convert back into a *trace-log*, the absolute values are calculated via propagation through the graph.

More formally, a *trace-DAG* is a triple (V, E^Θ, E^τ) where V is a set of vertices (one for each event in the *trace-log*) and E^Θ, E^τ are sets of directed dependency edges, *thread edges* and *time edges*, respectively. *Thread edges* are unlabelled and chain together successive events executed by a single thread; *time edges* are labelled with durations which express a *minimum* delay between two events. Two vertices connected by a thread edge are always connected by a time edge as well. There may also be time edges between concurrently dependent events (e.g. waiting on a task result), though the delay label will always be 0 for concurrent dependencies to indicate a wait (i.e. a "happens-before" edge).

Optimisations are estimated by editing the *trace-DAG*. All three optimisations described, *convert-to-inline*, *hoist-branch*, and *convert-to-combinator*, are based on the same core graph edit: moving the execution events of a task to the position of its spawn event.[4] To do this, the edges terminating and originating at the task's spawn event vertex are redirected to the task's execution start- and end-execution event vertices, respectively (Eq. 1). The edges that previously terminated at the task's start-execution event vertex and those originating at the task's end-execution event vertex are stitched together (Eq. 2). Formally, given task t, write t_p for its spawn event vertex (see Section 2), and t_s and t_e for its start- and end-execution event vertices. Write the replacement of edge(s) X with edge(s) Y as $X \mapsto Y$. Using this graph notation, and writing (v, w, d) for an edge from v to w with label d, the graph edit for inlining a task's execution is:

$$(v,\ t_\mathrm{p},\ d),\ (t_\mathrm{p},\ w,\ d') \mapsto (v,\ t_\mathrm{s},\ d),\ (t_\mathrm{e},\ w,\ d') \tag{1}$$

$$(v,\ t_\mathrm{s},\ d),\ (t_\mathrm{e},\ w,\ d') \mapsto (v,\ w,\ d + d') \tag{2}$$

[4]For *convert-to-combinator* doing this naively would over-sequentialise the sub-tasks in Fig. 3, but the inlined events from `longOperation1` and `longOperation2` are easily separated in a later phase which creates the detailed combinator expression.

(edits for thread edges are equivalent, sans labels.)

To derive a *trace-log* from a *trace-DAG*, the absolute timestamp and thread-id values for each vertex are propagated through the graph along the time- and thread-edges, respectively. These absolute values are then used to create an event for each vertex (the event type and auxiliary data is copied from the original event corresponding to the vertex). The derived events are then written (grouped by thread) in execution order as a new *trace-log*. While thread-ids are trivially propagated from the first vertex on a thread, the time-stamp for the event created for vertex w is calculated by:

$$\mathcal{T}(w) = \max_{(v,w,d)\in E^{\tau}} \mathcal{T}(v) + d \tag{3}$$

(this is naturally 0 if w has no predecessors.)

To improve estimation accuracy when *trace-DAGs* have been edited, thread sleeps are adjusted by modifying duration labels on related time edges before calculating the absolute time-stamps. Intuitively, this involves calculating the earliest time a task could begin executing based on when it was spawned and when the previous work on the thread finished. The thread's previous work completion time is determined by a *thread-waiting-for-task* event (one of the *thread-pool* scheduling events). The earliest a task can begin executing is based on the *task-spawn* event. Thus, given *thread-waiting-for-task* and *task-spawn* event nodes w and s, respectively, the (simplified[5]) calculation of the updated sleep duration is:

$$\mathcal{D}(s,w) = \max(0, \mathcal{T}(s) - \mathcal{T}(w)) \tag{4}$$

This value is assigned as the time-edge label between the *thread-waiting-for-task* and *thread-processing-task* event nodes (which determine the start and end of the sleep).

Sound by design *Rehype*'s re-ordering of the *trace-log* is sound, given adherence to the task concurrency model described in Section 2. That is, the re-ordered *trace-log* reflects the *trace-log* the program would generate given the immediate execution of the *task-body function*. Recall that the concurrency model assumes solely future-based concurrent communication. In particular, tasks intuitively correspond to (externally) pure functions and their behaviour is determined by their parameters. Thus, a task will behave the same whether it is executed immediately (as a direct call to the *task-body function*) or scheduled on a thread-pool; therefore re-ordering of task execution events in a *trace-log* is sound.

4.3 Multiple Improvements

Having described the analyser for estimating a single improvement, we now extend it to sets of improvements. Changes to a program's concurrency naturally interfere with each other. This means that the estimated effects of multiple improvements are not summative, that is, the effect of implementing them together is not equal to the sum of the effects of implementing them individually. As such, the final step of *Rehype* is to estimate the effect of each combination of improvements δ, to accurately estimate their overall effect. This can result in a combinatorial explosion of 2^n possible combinations, for n improvements.

Given the potential size of the combinatorial space, it is important to limit the space as much as possible and search it in a sensible manner. To limit the space, *Rehype* identifies sets of mutually exclusive improvements. To search the remaining space, *Rehype* uses a form of hill-climbing to more efficiently identify the most beneficial combinations. This hill-climbing optimises for a particular *composite* metric (Section 4.4) that minimises thread usage and execution duration, to maximise throughput.

Two improvements are mutually exclusive if they would require conflicting trace-log changes (and hence they would require conflicting source-level changes). In essence an improvement defines "do X to tasks Y", such as "do convert-to-inline to tasks A, B, and C". As an example, two mutually exclusive improvements might be: "do convert-to-combinator to tasks A and B" and "do convert-to-inline to tasks B and C". These improvements require different operations be applied to the same tasks (in this case, B), and are thus mutually exclusive.

Efficient methods for searching combinatorial space is a well established area of research and, in future work, we plan to investigate how these can be better leveraged in the improvement combinatorial space.

4.4 Measuring and Selecting Improvements

Rehype calculates various metrics from a *trace-log* for program p; these can be used to compare the original *trace-log* $[\![p]\!]$ with the *trace-log* $\beta_\delta\,[\![p]\!]$ incorporating the improvements δ and with the *trace-log* $[\![p_\delta]\!]$ from executing the modified program p_δ. Metrics include *time-series* metrics which give a separate value for each timestamp (e.g. number of threads in each state – active, waiting, and unoccupied) and *aggregate* metrics (e.g. wall-clock execution duration and summary statistics (e.g. mean, max)) derived from time-series metrics.

The thread-usage metric values can be plotted to aid identifying bottlenecks and spikes in concurrent thread usage. The aggregate metrics can be used individually or collectively to sort and select improvements, or combinations of improvements, that best optimise for the improvements the developer seeks. In the simplest case, sets of improvements along with their associated metric values can be inserted into a spreadsheet for the developer to analyse. *Derived metrics* can usefully combine multiple metrics into a new metric. For example, squaring and averaging a set of metric values can weight greater single-metric improvements more significantly.

5 EXPERIMENTAL RESULTS

We present evaluations of the accuracy of estimated *trace-logs* via plotted time-series metrics (Fig. 4) and the performance effects achieved by implementing the suggested improvements. The results demonstrate the effectiveness of *Rehype* for large real-world systems, the potential and validity of estimating the effect of improvements based on previous executions for concurrency performance analysis, and the significant performance improvements achievable by reducing concurrency.

We evaluate *Rehype* against a (proprietary) industrial Java API server (c. 500kLoC) for a consumer web and mobile application that: employs the task-based concurrency model (Section 2); has been in production for over 5 years; and has extensive automated testing (providing practical confirmation that the improvements preserve its functional results, as explained in Section 2). Measurements

[5]In practice, to ensure robustness, the sleep duration calculation is slightly more complex as it deals with a few (otherwise unimportant) nodes and time edges.

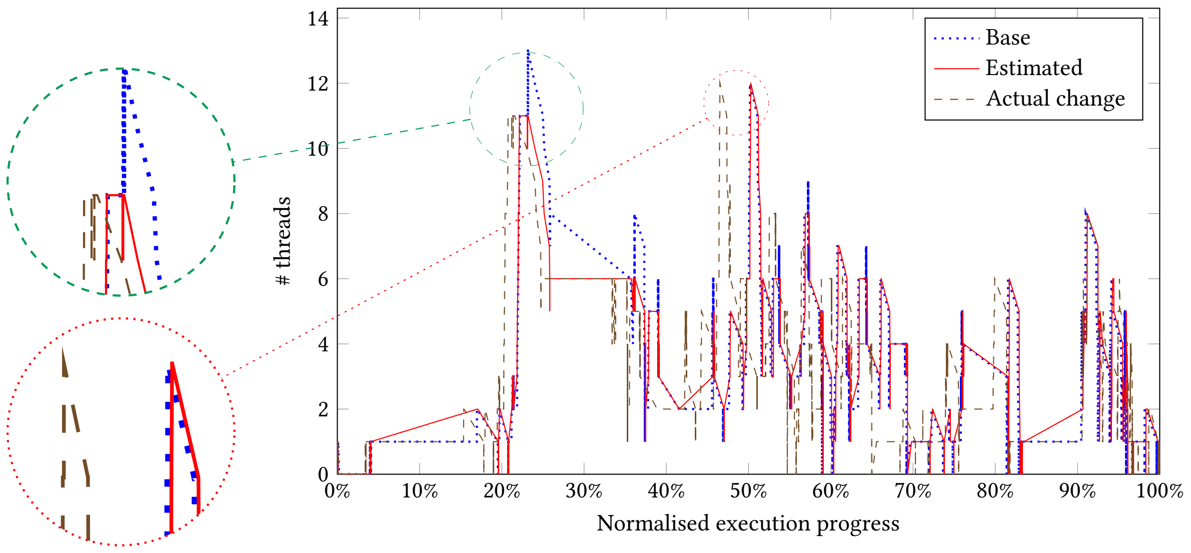

Figure 4: The concurrency profiles for the evaluated server on our synthetic workload – pre-change, estimated post-change and actual post-change.

are taken using a synthetic series-of-requests workload which enqueues a series of API requests to simulate standard use of the system. This provides a consistent workload to evaluate the effects of implemented improvements.

For completeness, all executions were performed with hardware specifications: 3.1GHz quad-core Intel Core i7 CPU, 16GB 2133 MHz LPDDR3 RAM, macOS 11.2.3, and software versions: Java SE 1.8.0_172, rustc 1.53.0-nightly (*Rehype* is implemented in Rust). However, these details are not especially relevant to the results as we focus on relative thread usage between executions. Multiple executions of the same program without changes had negligible differences in thread usage (the differences are orders of magnitude smaller than those caused by code changes).

We first evaluate estimation accuracy on both aggregate and time-series metrics with regards to a single suggested improvement to isolate its effects. In our evaluation we execute the original program once, then we estimate the effect of an improvement, and, finally, we implement the improvement and execute the program again. To assess estimation accuracy we compare a composite of the aggregate metrics between the estimate *trace-log* and a *trace-log* generated after implementing the improvement. The estimated and real-change *trace-logs'* composite metric values are within 0.5% of each other. More informatively, Fig. 4 contains the *concurrency profiles* of the *trace-logs*. The *concurrency profile* of a *trace-log* is the "shape" of a program's thread usage through an execution; this is drawn here as the number of *occupied* threads against normalised execution time. Multiple executions of a program should have similar concurrency profiles. As shown, the *estimated* and *actual change* profiles are very similar, including their deviations from the base profile. In Fig. 4 the top (green) dashed magnified circle highlights where the suggested improvement affected thread usage as well as illustrating the similarity between the *estimate* and *actual change* profiles. The bottom (red) dotted magnified circle shows an area unaffected by the improvement; all three profiles have a similar shape, despite minor temporal displacement.

Finally, *Rehype* identified 59 individual potential improvements to the server. By estimating over 10 000 improvement combinations, a combination of 19 improvements was identified as most beneficial (noting that some larger combinations produced less benefit). Combinations were first filtered with a requirement that they do not negatively impact execution duration or increase peak or average thread usage. The remaining combinations were then ranked using a composite metric, which combined execution duration and peak and average thread usage, to represent overall benefit. Implementing the selected combination of 19 improvements more than doubled the potential request-processing throughput of the evaluated server. Specifically, the peak and mean concurrent total thread usage dropped from 13 to 5 and 3.5 to 1.5, respectively, and the peak and mean concurrent waiting thread usage dropped from 9 to 3 and 2.1 to 0.3, respectively. As the server uses a standard thread pool of size 20, prior to the change two requests peaking in thread usage at the same time would saturate the thread pool and begin to throttle, whereas post change, the server could process four concurrent requests without over-saturating the thread pool (four concurrent requests would, at peak, only consume 20 threads). As the wall-clock request duration did not change statistically significantly between the pre- and post-change programs, the server gained the increased (doubled) concurrent processing capacity without compromising individual request speed.

6 RELATED WORK

There are two primary areas of related work: *performance prediction* and *concurrency analysis*. *Rehype* differs from most performance prediction work by focusing on concurrency and estimating improvements based on an existing trace. It differs from most concurrency analysis work in only using dynamic analysis and in its focus on performance rather than bug detection.

Performance prediction. Whereas *Rehype* performs estimation on an already captured execution trace, many existing approaches use some form of re-execution to predict performance. *Dynamic performance stubs* by Trapp [12] mock a module to have it exhibit certain performance characteristics (such as CPU usage or memory usage). The program is then executed to determine the potential performance improvement of optimising the targeted module. Sodhi et al. [11] describe *performance skeletons*, "synthetically generated short running program(s) whose execution time always reflects the performance of the application it represents" [11]. Skeletons capture the performance behaviour of a program by utilising system resources (CPU, memory, disk IO, and network IO) in relative proportion to the target program; they are automatically generated based on execution traces of the target program. Sodhi et al. report performance prediction accuracy of up to 95% using skeletons that execute for 5 to 10 seconds. Performance skeletons are used to predict the performance of a program in a new environment (e.g. system configuration, data centre cluster, etc).

Concurrency analysis. Whereas *Rehype* focuses on performance and is purely dynamic-analysis-based, much of the existing work on concurrency analysis focuses on detecting concurrency-based bugs such as race conditions [1, 2, 7], deadlocks [1], and more-general thread-safety violations [3, 6]. Most approaches are either based on static analysis [7], potentially with subsequent confirmation via dynamic analysis [14] or a hybrid approach [1], or use dynamic analysis such as execution-replay [10] or execution-modification [6] (such as injected delays) to trigger bugs. While there are some primarily dynamic analysis-based approaches (e.g. Chen et al. [2]), these also focus on bug detection. We hypothesise that concurrency analysis research currently focuses on bug detection for two reasons: 1) there is a clear, immediate, and measurable benefit – every additional bug detected is valuable; and 2) concurrency performance analysis is predicated on a low-overhead tracer that avoids distorting concurrent behaviour (whereas bug detection may use higher overhead dynamic analysis tools).

Finally, Moesus et al. [8] present a technique for ranking potential refactorings (such as using Java's `StringBuilder` to concatenate strings instead of the plus operator) based on estimating performance improvements. Their approach estimates the benefit of a given refactoring by combining the estimate execution frequency of a piece of code and the approximate performance improvement achievable by implementing the refactoring. *Rehype* differs both in terms of the type of refactorings and in the approach to estimating performance improvements. However, both share the base concept that for automated performance analysis to be useful it must accurately estimate the performance improvement of a refactoring.

7 DISCUSSION

Is this an artefact of Java's thread implementation? Theoretically, wait-limited tasks are problematic regardless of the implementation. In practice, runtimes that have lightweight threading systems (such as Go's) are impacted less by this problem than heavyweight thread systems (such as Java) that use operating system threads. If threads are "infinitely" scalable, then using some threads simply to wait is not necessarily problematic. Of course, lightweight threading systems introduce other issues, such as the scheduling overhead inherent in managing thousands of "threads" and potentially losing some CPU pre-emption benefits.

Could better implementation/developer practice avoid these problems? Yes; as discussed in Section 4, perfect use of combinators can result in wait-free task concurrency. However, wait-limited tasks are a real problem that exist in industrial programs today; it seems likely they will continue to be a problem in the future given the relative complexity of using combinators. Fundamentally, more-complex concurrency constructs can achieve better performance, but simpler constructs are more accessible and regularly used by developers. Improving the performance of these simpler constructs (such as by removing wait-limited tasks), can have a significant impact on real-world concurrency usage. In an ideal world, developers would be able to program using simpler constructs and achieve performance close to that achievable with more-complex constructs (or even convert to using the more-complex constructs where useful, as our convert-to-combinator optimisation does).

8 CONCLUSION

We have introduced *Rehype*, a concurrency performance analysis system that uses execution-trace analysis to propose source-level optimisations and estimate their performance effects. *Rehype* analyses task-based concurrency, and more specifically a future-based model of tasks. While *Rehype* consists of a tracer and an analyser, this paper focused on the analyser. The analyser transforms the generated *trace-log* into a *trace-DAG*, performs edits to reflect a source optimisation, derives a *trace-log* from the edited *trace-DAG*, and compares the performance-metric values of the estimated *trace-log* against the original. We evaluated *Rehype* on a substantial industrial API server and found the estimation to be highly accurate. Furthermore, improvements suggested by *Rehype* more than doubled the server's potential throughput.

The key idea of this paper is to estimate *quantifiable* performance effects to identify improvements, instead of detecting *potentially* inefficient patterns. While we focus on Java concurrency optimisations in this paper, the approach should be applicable to other languages and optimisations.

In the future we plan to develop more optimisations, methods for addressing the combinatorial explosion of improvements, and support for concurrency models. A companion paper describes *Scopda* [9] which considers the problem of generating git-style source code patches for the optimisations suggested by *Rehype*.

ACKNOWLEDGMENTS

We thank the referees for their helpful suggestions. The first author was funded by the Engineering and Physical Sciences Research Council (EPSRC), the Cambridge Trusts, and the University of Cambridge Department of Computer Science and Technology.

REFERENCES

[1] Luc Bläser. 2018. Practical detection of concurrency issues at coding time. In *the 27th ACM SIGSOFT International Symposium*. ACM Press, New York, New York, USA, 221–231. https://doi.org/10.1145/3213846.3213853

[2] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. 2019. Detecting Data Races Caused by Inconsistent Lock Protection in Device Drivers. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (March 2019), 366–376. https://doi.org/10.1109/SANER.2019.8668017

[3] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 266–277. https://doi.org/10.1109/ICSE.2017.32

[4] GitHub [n. d.]. GitHub. https://github.com. Accessed May 2021.

[5] Java Documentation 2021. Java 8 `java.util.concurrent` documentation. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html. Accessed May 2021.

[6] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection. In *the 27th ACM Symposium*. ACM Press, New York, New York, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[7] Yanze Li, Bozhen Liu, and Jeff Huang. 2019. SWORD: A Scalable Whole Program Race Detector for Java. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 75–78. https://doi.org/10.1109/ICSE-Companion.2019.00042

[8] Nikolai Moesus, Matthias Scholze, Sebastian Schlesinger, and Paula Herber. 2018. Automated Selection of Software Refactorings that Improve Performance. In *ICSOFT*.

[9] Indigo Orton and Alan Mycroft. 2021. Source Code Patches from Dynamic Analysis. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21)*. Association for Computing Machinery, 8. https://doi.org/10.1145/3464971.3468416

[10] Ernest Pobee, Xiupei Mei, and W K Chan. 2019. Efficient Transaction-Based Deterministic Replay for Multi-threaded Programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 760–771. https://doi.org/10.1109/ASE.2019.00076

[11] Sukhdeep Sodhi, Jaspal Subhlok, and Qiang Xu. 2008. Performance prediction with skeletons. *Cluster Computing* 11, 2 (2008), 151–165. https://doi.org/10.1007/s10586-007-0039-2

[12] Peter Trapp. 2011. *Performance Improvements Using Dynamic Performance Stubs*. Ph.D. Dissertation. De Montfort University.

[13] R G Urma, M Fusco, and A Mycroft. 2018. Modern Java in Action: Lambdas, streams, functional and reactive programming.

[14] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs through Sequential Errors. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 251–264. https://doi.org/10.1145/1961295.1950395