Implicit invocation meets safe, implicit concurrency

Yuheng Long  
_Iowa State University_

Sean L. Mooney  
_Iowa State University_

Tyler Sondag  
_Iowa State University_

Hridesh Rajan  
_Iowa State University, hridesh@iastate.edu_

Follow this and additional works at: [http://lib.dr.iastate.edu/cs_conf](http://lib.dr.iastate.edu/cs_conf)  
Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

**Recommended Citation**  
[http://lib.dr.iastate.edu/cs_conf/40](http://lib.dr.iastate.edu/cs_conf/40)
Implicit invocation meets safe, implicit concurrency

Abstract
Writing correct and efficient concurrent programs still remains a challenge. Explicit concurrency is difficult, error prone, and creates code which is hard to maintain and debug. This type of concurrency also treats modular program design and concurrency as separate goals, where modularity often suffers. To solve these problems, we are designing a new language that we call Panini. In this paper, we focus on Panini's asynchronous, typed events which reconcile the modularity goal promoted by the implicit invocation design style with the concurrency goal of exposing potential concurrency between the execution of subjects and observers.

Since modularity is improved and concurrency is implicit in Panini, programs are easier to reason about and maintain. The language incorporates a static analysis to determine potential conflicts between handlers and a dynamic analysis which uses the conflict information to determine a safe order for handler invocation. This mechanism avoids races and deadlocks entirely, yielding programs with a guaranteed deterministic semantics. To evaluate our language design and implementation we show several examples of its usage as well as an empirical study of program performance. We found that not only is developing and understanding Panini programs significantly easier compared to standard concurrent object-oriented programs, but also performance of Panini programs is comparable to their equivalent hand-tuned versions written using Java's fork-join framework.

Disciplines
Computer Sciences | Programming Languages and Compilers | Software Engineering

Comments

Rights

This conference proceeding is available at Iowa State University Digital Repository: http://lib.dr.iastate.edu/cs_conf/40
Implicit Invocation Meets Safe, Implicit Concurrency

Yuheng Long  Sean L. Mooney  Tyler Sondag  Hridesh Rajan

Dept. of Computer Science, Iowa State University
{csgzlong,smooney,sondag,hridesh@iastate.edu}

Abstract

Writing correct and efficient concurrent programs still remains a challenge. Explicit concurrency is difficult, error prone, and creates code which is hard to maintain and debug. This type of concurrency also treats modular program design and concurrency as separate goals, where modularity often suffers. To solve these problems, we are designing a new language that we call Pānini. In this paper, we focus on Pānini’s asynchronous, typed events which reconcile the modularity goal promoted by the implicit invocation design style with the concurrency goal of exposing potential concurrency between the execution of subjects and observers. Since modularity is improved and concurrency is implicit in Pānini, programs are easier to reason about and maintain. The language incorporates a static analysis to determine potential conflicts between handlers and a dynamic analysis which uses the conflict information to determine a safe order for handler invocation. This mechanism avoids races and deadlocks entirely, yielding programs with a guaranteed deterministic semantics. To evaluate our language design and implementation we show several examples of its usage as well as an empirical study of program performance. We found that not only is developing and understanding Pānini programs significantly easier compared to standard concurrent object-oriented programs, but also performance of Pānini programs is comparable to their equivalent hand-tuned versions written using Java’s fork-join framework.

Categories and Subject Descriptors D.3.2 [Language Classifications]: Concurrent, distributed, and parallel languages; D.3.3 [Language Constructs and Features]: Concurrent programming structures, Patterns

General Terms Languages, Design, Performance

Keywords Safe Implicit Concurrency, Modularity

1. Introduction

The idea behind Pānini’s design is that if programmers structure their system to improve modularity in its design, they should get concurrency for free.

1.1 Explicit Concurrency Features

It is widely accepted that multicore computing is becoming the norm. However, writing correct and efficient concurrent programs using concurrency-unsafe features remains a challenge [4, 28, 30, 45]. A language feature is concurrency-unsafe if its usage may give rise to program execution sequences containing two or more memory accesses to the same location that are not ordered by a happens-before relation [19]. Several such language features exist in common language libraries. For example, threads, Futures, and FutureTasks are all included in the Java programming language’s standard library [20, 25]. Using such libraries has advantages, e.g., they can encapsulate complex synchronization code and allow its reuse. However, their main disadvantage is that today they do not provide guarantees such as race freedom, deadlock freedom and sequential semantics. This makes it much harder and error prone to write correct concurrent programs.

To illustrate, consider the implementation of a genetic algorithm in Java presented in Figure 1. The idea behind a genetic algorithm is to mimic the process of natural selection. Genetic algorithms are computationally intensive and are useful for many optimization problems [39]. The main concept is that searching for a desirable state is done by combining two parent states instead of modifying a single state [39]. An initial generation with \( n \) members is given to the algorithm. Next, a crossover function is used to combine different members of the generation in order to develop the next generation (lines 10–16 in Figure 1). Optionally, members of the offspring may randomly be mutated slightly (lines 18–23 in Figure 1). Finally, members of the generation (or an entire generation) are ranked using a fitness function (lines 25–29 in Figure 1).

Multiple Concerns of the Genetic Algorithm. In the OO implementation of the genetic algorithm in Figure 1, there are three concerns standard to the genetic algorithm: crossover (creating a new generation), mutation (random changes to children), and fitness calculation (how good is the new generation). Logging of each generation is another concern added here, since it may be desirable to observe the space searched by the algorithm (lines 17 and 24). The final concern in the example is concurrency (lines 4, 7–9, and 30–33). In this example, production of a generation is run as a FutureTask, but other solutions are also possible. The shading represents different concerns as illustrated in the legend.

1.2 Problems with Explicit Concurrency Features

Explicit concurrency. With explicit concurrency, programmers must divide the program into independent tasks. Next, they must handle creating and managing threads. A problem with the concurrency-unsafe language features described above and illustrated in Figure 1 is that correctness is difficult to ensure since it relies on all objects obeying a usage policy [20]. Since such policies cannot automatically be enforced by a library based approach [20], the burden on the programmers is increased and errors arise (e.g., deadlock, races, etc.). Also, the non-determinism introduced by such mechanisms makes debugging hard since errors are difficult to reproduce [33]. Furthermore, this style of explicit parallelism can hurt the design and maintainability of the resulting code [37].
Separation of modular and concurrent design. Another shortcoming of these language features, or perhaps the discipline that they promote, is that they treat modular program design and concurrent program design as two separate and orthogonal goals.

From a quick glance at Figure 1, it is quite clear that the five concerns are tangled. For example, the code for concurrency (lines 4, 7-9, and 30-33) is interleaved with the logic of the algorithm (the other four concerns). Also, the code for logging occurs in two separate places (lines 17 and 24). This arises from implementing a standard well understood sequential approach and then afterward attempting to expose concurrency rather than pursuing modularity and concurrency simultaneously. Aside from this code having poor modularity, it is not immediately clear if there is any potential concurrency between the individual concerns (crossover, mutation, logging, and fitness calculation).

1.3 Contributions

Our language, Pānini, addresses these problems. The key idea behind Pānini’s design is to provide programmers with mechanisms to utilize prevalent idioms in modular program design. These mechanisms for modularity in turn automatically provide concurrency in a safe, predictable manner. This paper discusses the notion of asynchronous, typed events of parallelism.

In Pānini, an event type appears on lines 1–3, whose name is GenAvailable and that declares one context variable g of type Generation on line 2. Context variables define the reflective information available at events of that type.

Certain classes, which we refer to as subjects from here onward, can provide methods, called handlers that are invoked (implicitly and potentially concurrently) when events are announced. The listing in Figure 2 has several examples of observers: CrossOver, Mutation, Logger and Fittest. A class can act as both subject and observer. For example, the classes CrossOver and Mutation are both subjects and observers for events of type GenAvailable.

In Pānini classes statically express (potential) interest in an event by providing a binding declaration. For example, the Mutate concern (lines 20-35) wants to randomly change some of the
population after it is created. So in the implementation of class Mutation there is a binding declaration (line 26) that says to run the method mutate (lines 27-35) when events of type GenAvailable are announced.

At runtime, these interests in events can be made concrete using the register statements. The class Mutation has a constructor on lines 22–25 that when called registers the current instance this to listen for events. After registration, when any event of type GenAvailable is announced the method mutate (lines 27-35) is run with the registered instance this as the receiver object.

Concurrently, the method logit (line 39) in class Logger will log each generation and the method check in class Fittest (lines 41-51) will determine the better fitness between the announced generation and the previously optimal generation.

Benefits of Panini’s Implementation. At a quick glance, we can see from the shading that the four remaining concerns are no longer tangled and they are separated into individual modules. This separation not only makes reasoning about their behavior simple but also allows us to expose potential concurrency between them.

Furthermore, the concurrency concern has been removed entirely since Panini’s implementation encapsulates concurrency management code. By not requiring users to write this code, Panini avoids any threat of incorrect or non-deterministic concurrency, thus easing the burden on programmers. This allows them to focus on creating a good, maintainable modular design.

Finally, additional concurrency between these four modules is now automatically exposed. Thus, Panini reconciles modular program design and concurrent program design.

Advantages of Panini’s Design Over Related Ideas. Panini is most similar to our previous work on Ptolemy [33], but Panini’s event types also expose concurrency advantages. Compared to similar ideas for aspect-oriented advice presented by Ansaloni et al. [2], Panini only exposes concurrency safely.

It is also similar to implicit invocation (II) languages [8,26] that also see events as a decoupling mechanism. The advantage of using Panini over an II language is that asynchronous, typed events in Panini allow developers to take advantage of the decoupling of subjects and observers to expose potential concurrency between their execution. A detailed comparison is presented in Section 6.

Panini also relies programmers from the burden of explicitly creating and maintaining threads, managing locks and shared memory. Thus it avoids the burden of reasoning about the usage of locks, which has several benefits. First, incorrect use of locks may have safety problems. Second, locks may degrade performance since acquiring and releasing a lock has overhead. Third, threads are cooperatively managed by Panini’s runtime, thus thrashing due to excessive threading is avoided. These benefits make Panini an interesting point in the design space of concurrent languages.

In summary, this work makes the following contributions:

1. Panini’s language design that reconciles implicit-invocation design style and implicit concurrency and provides a simple and flexible concurrency model such that Panini programs are
   - free of data races,
   - free of deadlocks, and
   - have a guaranteed deterministic semantics—a given input is always expected to produce the same output [31];
2. an efficient implementation of Panini’s design as an extension of the JastAdd compiler [9] that relies on:
   - an algorithm for finding inter-handler dependence at registration time to maximize concurrency,
   - a simple and efficient algorithm for scheduling concurrent tasks that builds on the fork/join framework [21];
3. a detailed analysis of Panini and closely related ideas;
4. and, an empirical performance analysis using canonical concurrency examples implemented using Panini and using standard techniques which shows that the performance and scalability of the implementations are comparable.

Overview. Next we describe Panini’s design. Section 2 describes Panini’s compiler and runtime system. Section 3 describes our performance evaluation and experimental results. Section 5 gives more examples in Panini. Finally, Section 6 surveys related work, and Section 7 describes future directions and concludes.

2. Panini’s Design

Panini, fl. c.400 BC,

Indian grammarian, known for his formulation of the Sanskrit grammar rules, the earliest work on linguistics.

In this section, we describe Panini’s design. Panini’s design builds on our previous work on the Ptolemy [33] and Eos [35] languages as well as implicitly parallel languages such as Jade [37]. Panini achieves concurrent speedup by executing handler methods concurrently. The novel features of Panini are found in its concurrency model and conflict-detection scheme. We do not present a formal semantics of Panini in this book, but interested readers may find it in our technical report [24].

2.1 Panini’s Syntax

Panini extends Java [16] with new mechanisms for declaring events and for announcing these events. These features are inspired by implicit invocation (II) languages such as RapidE [8] and our previous work on Ptolemy [33]. These syntax extensions are shown in Figure 3. In this figure, productions of the form “...” represent all existing rules for Java [16] plus rules on the right.

Figure 3. Panini’s syntax extensions to Java.

In this syntax, the novel features are: event type declarations (event), event announcement statements (announce), and handler registration statements (register). Since Panini is an implicitly concurrent language, it does not feature any construct for spawning threads or for mutually exclusive access to shared memory. Rather, concurrent execution is facilitated by announcing events, using the announce statement, which may cause handlers to run concurrently. Examples of the syntax can be seen in Figure 2. This example is described thoroughly in Section 1.3.

Top-level Declarations. Class, interface and enum declarations are the same as in Java and not shown. We add a new declaration for events. An event type declaration (EventDec) has a name (Identifier), and zero or more context variable declarations (ContextVariable*). These context declarations specify the types and names of reflective information exposed by conforming events. An example is given in Figure 2 on lines 1-3 where event GenAvailable has one context variable Generation that denotes the generation which is now available. The intention of this event type declaration is to provide a named abstraction for a set of events that result from a generation being ready.
Like Eos [34,36], classes in Pānini may also contain binding declarations. A binding declaration (\(<\text{BindingDecl}\>)\) mainly consists of two parts: an event type name (\(<\text{Type}\>)\) and a method name (\(<\text{Identifier}\>)\). For example, in Figure 3 on line 10 the class CrossOver declares a binding such that the cross method is invoked whenever an event of type GenAvailable is announced. We call such methods handler methods and they may run concurrently with other handler methods for the same event.

Pānini’s New Statements. Pānini has all the standard object-oriented expressions and statements as in Java. New to Pānini is the registration statement (\(<\text{RegisterStmt}\>)\) and (\(<\text{AnnounceStmt}\>)\). Like II languages and Ptolemy [33], a module in Pānini can express interest in events, e.g., to implement the observer design pattern [13]. Just like II languages, where one has to write an statement for registering a handler with each event in a set, and similar to Ptolemy [33], such modules run registration statements. Examples are shown on lines 7, 23, 38 and 44 in Figure 2. The example on line 7 registers the this object to receive notification when events of type GenAvailable are signaled.

2.2Concurrency in Pānini

The announce statement enables concurrency in Pānini. The statement announce p (\(<\text{Expr}\>)\); signals an event of type p, which may run any handler methods that are applicable to p asynchronously, and waits for the handlers to finish. In Figure 2, the body of the cross method contains an announce statement on line 18. On evaluation of the announce statement, Pānini first looks for any applicable handlers. Here, the handlers CrossOver, Mutation, Logger, and Fittest, are declared to handle the events of type GenAvailable. Such handlers may run concurrently, depending on whether they interfere with each other.

The evaluation of the announce statement then continues with evaluating the sequence on line 18, which returns from the method. The announcement of the event allows for potential concurrent execution of the bodies of the cross (lines 11–19), mutate (lines 27–35), logit (line 38), and check (lines 45–51) methods.

The announce statement also binds values to the event type declaration’s context variables. For example, when announcing event GenAvailable on line 18, g1 is bound to the context variable g on line 2. This binding makes the new generation available in the context variable g, which is needed by the context declared for the event type GenAvailable.

2.3 Pānini’s Handler Conflict Detection Scheme

Pānini uses static effect computation [44] and a dynamic conflict detection scheme to compute a schedule for execution of handlers that maximizes concurrency while ensuring a deterministic semantics of programs. This is similar to Jade [37], where the implementation tries to discover concurrency. But unlike Jade, we do not require effect annotations. Pānini’s compiler generates code to compute the potential effect of all handlers. At runtime, when a handler registers with an event, Pānini’s runtime uses these statically computed effects to decide the execution schedule of handlers.

Effects of a Method. The effects of a method are modeled as a set that may contain four kinds of effects: 1) read effect: a class and its field that may be read; 2) write effect: a class and its field that may be written; 3) announce effect: an event that may be announced by the method; 4) register effect: whether this method may evaluate a register statement. These sets are generated for each method in the program and inserted in the generated code as synthetic methods. For library methods, their effects are computed by analyzing their bytecode and inserted directly at call-sites.

Detecting Dependencies between Handlers. When a register statement is run with a handler as argument, dependence between this handler and already registered handlers for that event is computed by comparing their effects. Two handlers may have read-write, write-write or register-announce dependencies.

Suppose the currently registering handler is \(h_a\) and \(h_i\) is in the sequence of already registered handlers. Handlers \(h_a\) and \(h_i\) may be register-announce dependent if \(h_a\) announces an event for which \(h_i\) registers a handler or vice versa. The handler \(h_a\) is read-write dependent on \(h_i\) if \(h_i\)’s reads conflict with \(h_a\)’s writes, or \(h_i\)’s writes conflict with \(h_a\)’s reads or writes. Two effect sets conflict, if they share an element. That is because, in the deterministic semantics, \(h_i\) should view the changes by \(h_i\), while \(h_i\)’s changes are invisible to \(h_i\), neither should the changes of \(h_i\) be overwritten by the changes of \(h_a\). We illustrate via an example in Figures 4–6.

Figure 4. Assume there are three handlers for the event type Ev in the program. At this point, none have registered yet. The registration order of handlers is \(A\) followed by \(B\) followed by \(C\). All four kinds of effects are shown for each handler.

In Figure 4 handler \(A\) reads the field balance of the class Account and handler \(C\) may write to the field balance. Since handler \(A\) registers earlier than handler \(C\), handler \(C\)’s writes conflict with handler \(A\)’s reads, as discussed above. Notice that a handler \(h\) could also announce an event, say \(p\). Then the read/write set of \(h\) could be enlarged over time, because new handlers for \(p\) may register later and the effects of these new handlers should propagate to \(h\). Pānini does these updates automatically when new handlers register for a certain event. To enable this, subjects are formed into a list for an event. Thus, when a handler registers, its changes are passed to these subjects, and these subjects merge the changes and recursively pass changes to other events when necessary. This continues until a fixpoint is reached (no more effects are added to the subjects). For example, in Figure 5 notice that handler \(A\) may announce events of type Ev. Thus after handler \(B\) registers, the effect set of handler \(A\) becomes the union of effect sets of handlers \(A\) and \(B\).

Figure 5. Effects after handler \(A\) and handler \(B\) have registered.

Finally, in Figure 6 the effect set of handler \(A\) becomes the union of effect sets of all the three handlers.

Figure 6. Effects after all three handlers have registered.

Handlers’ Hierarchy. Pānini groups handlers into hierarchies, based on handler dependencies. In the first level of the hierarchy, none of the handlers have a dependency on any other handlers, while any handler in the second level depends on a subset of the handlers in the first level and no other handlers. For example, handler \(C\) conflicts with handler \(A\) (discussed previously). Similarly, handlers in the third level may depend on handlers in the first two levels, but no handlers in any other level. It is possible that the effects of one handler will become larger (mentioned above) and
in response to this, Pāṇini will reorder the hierarchy dynamically. Thus, the example above will have a two level hierarchy, with handlers $A$ and $B$ in the first level, while, handler $C$ in the second.

**Event Registration.** When a handler, say $h$, registers with event $p$, we first propagate its effects to the subjects of $p$, then the dependencies between $h$ and the previous registered handlers are computed based on the effect set. After dependencies are calculated, the handler is put into a proper level of the hierarchy. In Figure 5 and Figure 6, since, handler $A$ may announce event type $Ev$, the effect sets of handler $B$ and handler $C$ are propagated to handler $A$ (as a subject). Because handler $B$ does not depend on handler $A$ (notice that read effects of the same field have no conflict), it is put in the first level. Since handler $C$'s writes conflict with handler $A$'s reads, it is put in the second level.

**Event Announcement and Task Scheduling Algorithm.** When a subject signals an event, Pāṇini executes the handlers in the first level concurrently (the subject itself blocks until all handlers are finished). After all the handlers in this level are done, handlers in the next level are released and run in parallel until all the handlers are finished. For example, since handlers $A$ and $B$ are both in the first level, they will run in parallel. Once they are completed, handler $C$ will run. If any of the handlers also announce an event, the handlers for that event will be scheduled, according to their conflict sets. Announce statements do not return until after all the handlers associated with the event are finished. This ensures correct synchronization for any state changes made by the handlers.

The computation of the dependency and the effect propagation is done when handlers register, based on the assumption that in a program, the number of announcements considerably outweighs the number of registrations. Therefore, the overhead of effect analysis is amortized over event announcements.

### 2.4 Properties of Pāṇini’s Design

Pāṇini does not have locks so it is deadlock free. It uses automatic conflict detection that ensures race freedom and guaranteed deterministic semantics. Our report has formal details and proofs of these properties. Its design, does not offer these guarantees if programmers use explicit locking and threads in the underlying Java language in a manner that creates deadlocks and data races.

### 3. Pāṇini’s Compiler and Runtime System

To a certain extent, implementing Pāṇini as a library is feasible. However, to get deadlock and race freedom and a deterministic semantics, which is crucial for writing correct and efficient concurrent programs, programmers will need to write extensive effect annotations (like Jade). This could be tedious and error prone so we implemented a compiler for Pāṇini using the JastAdd extensible compiler system. Its compiler and associated examples are available for download from [http://panini.j.org](http://panini.j.org).

As its backend, Pāṇini’s runtime system uses the fork/join framework. This framework uses the work stealing algorithm and works well for recursive algorithms. We observed that handlers usually also act as subjects and recursively announce events, thus Pāṇini was built based on this framework. When an event is announced by a publisher, all handlers that are applicable are wrapped and put into the framework and may execute concurrently. Below we describe key parts of our implementation strategy.

**Event type.** An event type declaration is transformed into an interface (an example is shown in Figure 7). A getter method is generated for each context variable of the event (Generation $g()$ on line 2 in Figure 7) so that the handlers can use this method to access the context variables. Two interfaces, namely $EventHandler$ and $EventPublisher$.

```java
public interface GenAvailable {
    public Generation g(); // An accessor for each context variable
}
public interface EventHandler extends IEventHandler {
    public void changedHandle(Generation g);
}
public interface EventPublisher extends IEventPublisher {
    public class EventFrame implements GenAvailable {
        public static void register(IEventHandler handler) {
            // Check whether this handler has registered before,
            // If yes return (no duplicate registration)
            // Or analyze the effects of the handler
            // Insert it into the handler hierarchy
        }
        public static void announce(GenAvailable ev) {
            // Iterate over registered handlers for the event wrapping
            // Them inside instances of tasks for concurrent execution.
            PaniniTask.colInvoke(tasks);
        }
    }
}
public static class GenAvailableTask extends PaniniTask {
    // Other helper methods elided
}
```

Figure 7. An event type is translated into an interface. Snippets from translation of event $GenAvailable$ in Figure 2 (lines 3–5) and $EventPublisher$ (line 6), are to be used by an inner class $EventFrame$ (lines 7–20), which hosts the register and announce methods for that event. Any class that has a binding declaration is instrumented to implement the $EventHandler$ interface, while any class that may announce is instrumented to implement the $EventPublisher$ interface.

**Event Announcement.** When a subject signals an event, the announces method (line 14 in Figure 7) is called. This method iterates over the handlers and executes all non-conflicting handlers as discussed in Section 2.3. The class $EventFrame$ uses a helper class (here $GenAvailableTask$ on line 21), to wrap the handlers (if any) before submitting them for execution.

**Handler Registration.** A register method is added to every class that has event bindings. First this method computes the effects of the handler. Next, this method registers to the named events in the class by calling the register method (lines 8–13 in Figure 7). This method will first check whether the current registering handler is already in the handler hierarchy to ensure no duplicate registration. Then the effects of the newly registered handler are compared against other previously registered handlers to calculate the dependency set of this handler (as discussed in Section 2.3). Finally, the handler is put into a proper level in the hierarchy.

### 4. Evaluation

We now evaluate the design and performance benefits of Pāṇini. All experiments were run on a system with a total of 12 cores (two 6-core AMD Opteron 2431 chips) running Fedora GNU/Linux.

**4.1 Analysis of Modularity and Concurrency Synergy**

Our goal is to analyze “if a program is modularized using Pāṇini does that also expose potential concurrency in its execution?”

We have already presented one such case in Section 3 where modularization of various concerns in the implementation of a genetic algorithm exposed potential concurrency between these concerns. We now analyze the speedup of the genetic algorithm implementations presented in Figure 1 and Figure 2. Recall that the first version is implemented by taking the sequential version and retrofitting it with thread and synchronization primitives, whereas the second version is implemented by modularizing the code. We first compared these implementations head-to-head. The results for this comparison are shown as black bars in Figure 8.

In this experiment, the average speedup over ten runs was taken with a generation (or population) size of 3000 and a depth (number
of generations) of 10. For a variety of generation sizes (1000–3000) and depths (8–11), speedups were similar.

The results show that Pà­ñini’s implementation achieved between 1 and 4x speedup for varying number of threads. This was quite surprising as we expected the concurrent version in Figure 4 to match or exceed the performance of Pà­ñini’s version since the OO version does not incur the overhead of implicit concurrency.

A careful analysis by a seasoned concurrent programmer revealed two problems with this seemingly straightforward concurrent code in Figure 4. Our expert pointed out: “the entire genetic algorithm code is wrapped in a future task. The method then submits the future task on line 30 and immediately invokes the method get, which limits concurrency. Furthermore, the compute() method calls (on line 16 and 23) are synchronous method calls, and thus, the two subtasks could not be run concurrently. As a result, the algorithm execution proceeds as a depth-first search tree (the right subtree will not be executed until the left subtree is done) but the intention is to execute the branches of the search tree concurrently.”

This analysis was both shocking and pleasant. Shocking in the sense that even with a relatively simple piece of concurrent code, correctness and efficiency was hard to get. Pleasant in the sense that the Pà­ñini code automatically dealt with these problems.

Following our concurrency expert’s advice, we created a second version of the object-oriented genetic algorithm using the fork/join framework [21]. The performance results of this “expert version” is shown in Figure 5 as gray bars. This figure shows that the speedups between the “expert version” and the Pà­ñini versions for this genetic algorithm are comparable.

In summary, our performance evaluation revealed correctness and efficiency problems with a relatively straightforward OO parallelization of the genetic algorithm, whereas Pà­ñini’s implementation didn’t have these problems. Fixing the problems with OO implementation by an expert led to comparable performance between implicit concurrency exposed by Pà­ñini and explicitly tuned concurrency exposed using the fork/join framework [21].

4.2 Performance Evaluation

The goal of this section is to analyze “how well do the Pà­ñini programs perform compared to a hand-tuned concurrent implementation of equivalent functionality?” We first describe our experimental setup and then analyze speedup realized by Pà­ñini’s implementation as well as the overheads.

4.2.1 Concurrency Benchmark Selection

To avoid bias and subtle concurrency problems similar to Section 4.1, we picked already implemented concurrent solutions of five computationally intensive kernels: Euler number, FFT, Fibonacci, integrate, and merge sort. Hand-tuned implementations of these kernels were already available [21].

Each program takes an input to vary the size of the workload (Euler: number of rows, FFT: size of matrix 2ⁿ, Fibonacci: xⁿ, Fibonacci number, integrate: number of exponents, and merge sort: array size 2ⁿ) For each example program, a sequential version was tested as well as concurrent versions ranging from 1 to 14 threads. Furthermore, three concurrent versions were tested:

1. an implementation using the fork/join framework [21],
2. a Pà­ñini version with no conflict between handlers, and
3. a second Pà­ñini’s implementation that was intentionally designed to have conflicts between handlers.

To introduce conflicts, we add another handler that aggregates the results of concurrently executing handlers. Thus, the third handler must wait for the other handlers to complete since it depends on them. For example, calculating a Fibonacci number, fib(n), is done by recursively calculating two subproblems, fib(n - 1) and fib(n - 2). With the fork/join framework, each of these subproblems is done by a separate task. When both of these tasks are completed, the spawning task adds them together. For Pà­ñini, each of these subproblems is handled in separate handlers. In the case with no conflicts, these are the only two handlers. In the case with conflicts, a third handler takes the result of the two handlers for the subproblems and adds them together.

4.2.2 Speedup over Sequential Implementation

Figure 9 shows a summary comparison of speedup between the three versions. In this figure, the average speedup across all five benchmarks was taken. For each program, large input sets were used (Euler: 39, FFT: 24, Fibonacci: 55, integrate: 7, and merge sort: 25). The line in the figure represents optimal speedup.

![Figure 9. Average speedup compared to sequential version across all benchmarks for varying number of threads. The line represents perfect scaling. This shows that Pà­ñini’s implementation scales similarly to hand-written fork/join implementation.](image-url)

This figure shows that the speedups between the three styles are comparable. Speedups for fork/join and Pà­ñini without conflicts are nearly the same. A statistical analysis showed that for all benchmarks, we do not see a statistically significant difference (p > 0.05) between fork/join and Pà­ñini with no conflicts.

From the figure, we can also see that Pà­ñini with conflicts has slightly lower speedup than both fork/join and Pà­ñini without conflicts, however, this decrease is rather small (average 6.5% decrease from fork/join). Note that since we are using a machine with 12 cores, performance levels drop off at 12 threads.

4.2.3 Overhead over the Sequential Implementation

We also measured the overhead involved with Pà­ñini as compared to the standard fork/join model. We first consider the average overhead across all benchmarks as shown in Figure 10.

![Figure 10. Overhead comparison between Pà­ñini and fork/join implementations.](image-url)
computed by determining the increase in runtime from the sequential version to the concurrent version with a single thread. For this experiment, we used large input sizes.

This figure shows us that while Pānini increases the overhead over fork/join, it is not a prohibitive amount. For example, for Pānini with no conflicts, we only see a 7.7% increase in overhead.

The implementation of this technique usually suffers from the problem of code-tangling: implementations of different concerns (i.e., transformation tasks) are all mixed together.

![Figure 10](image-url) Average overhead compared to sequential version across all benchmarks for each technique.

In the rest of this section, we present three examples.

### 5. Other Examples in Pānini

To further assess Pānini’s ability to achieve a synergy between modularity and concurrency goals, we have implemented several representative examples and they worked out beautifully. In the rest of this section, we present three examples.

**Concurrency in Compiler Implementations.** In the art of writing compilers, performance often has higher priority than modularity. Compiler designers employ all kinds of techniques to optimize their compilers. For example, merging transformation passes which perform different transformation tasks in the same traversal, is a common practice in writing multi-pass compilers. However, the implementation of this technique usually suffers from the problem of code-tangling: implementations of different concerns (i.e., transformation tasks) are all mixed together.

![Figure 11](image-url) Average overhead for Fibonacci benchmark for varying input size and each scheduling strategy.

Figure 11 shows a summary comparison of overhead as program input size changes. In this figure, the overhead for the Fibonacci program is shown with a variety of input sizes. Again, overhead is calculated by determining the increase in runtime from the sequential version to the concurrent version with a single thread.

This figure shows that as input size increases, overhead decreases. Here, overhead decreases to as low as 5.5% additional overhead for Pānini with no conflicts. Pānini with conflicts only incurs an additional 1.2% overhead for larger input sizes. Each of the differences in overhead (fork/join vs Pānini without conflicts, fork/join vs Pānini with conflicts, and Pānini with vs Pānini without conflicts) was always statistically significant ($p < 0.05$).

**Figure 12. Snippets of an AST with an Effects System**

Figure 12 illustrates this via snippets from an abstract syntax tree (AST). It shows concerns for method declarations, expressions, and two concrete expressions: a sequence expression ($e_1; e_2$) and a field get expression ($e.f$). As an example compiler pass, we show computation of effects for these AST nodes. The effect computation concern is scattered and tangled with the AST nodes. This is a common problem in compiler design where the abstract syntax tree hierarchy imposes a modularization based on language features whereas compiler developers may also want another modularization based on passes, e.g., type checking, error reporting, code generation, etc. The visitor design pattern solves this problem to a certain extent but it has other problems.

![Figure 13](image-url) Pānini’s version of visiting an abstract syntax tree.
Pañini handles this modularization problem readily as shown in Figure 15. In this implementation, we introduce a method visit in each AST node. This method recursively visits the children of the node. At the same time, it announces events corresponding to the AST node. For example, a method declaration announces an event of type MethodVisited declared on line 1 and announced on line 8. Similarly, the AST node sequence expression and field set expression announce events of type SequenceVisited and FieldGetVisited on lines 16 and 22 respectively.

The implementation of the effect concern is modularized as the class ComputeEffect. This class has two bindings that say to run the method start when an event of type MethodVisited is announced and add when an event of type FieldGetVisited is announced. The constructor for this class registers itself to receive event announcements and initializes a hashtable to store effects per method. The method add inserts a read effect in this hashtable corresponding to the entry of the current method.

This Pañini program manifests a few design advantages. First, the AST implementation is completely separated from effect analysis. Also, unlike the visitor pattern, the ComputeEffect class need not implement a default functionality for all AST nodes. Furthermore, other passes such as type checking, error reporting, code generation, etc can also reuse the AST events.

Modular and Concurrent Image Processing. This example is adapted from and inspired by the ImageJ image processing toolkit [17]. For simplicity, assume that this library uses a class List and Hashtable similar to the classes in the java.util package. We have also omitted the irrelevant initializations of these classes. The class Image (lines 24–29) maintains a list of pixels. The method set for this class (lines 27–29) sets the value of a pixel at a given location to the specified integer value.

An example requirement for such a collection could be to signal changes of elements as an event. Other components may be interested in such events, e.g., for implementing incremental functionalities which rely on analyzing the increments. One such requirement for a list of pixels is to incrementally compute the Non-parametric Histogram-Base Thresholding [15]. Thresholding is a method for image segmentation that is typically used to identify objects in an image. The threshold functionality may not be useful for all applications that use the image class, thus it would be sensible to keep its implementation separate from the image class to maximize reuse of the image class. Figure 14 shows the implementation of two thresholding methods in classes Percentile and GlasbeyThreshold. Pañini’s implementation allows the threshold computation concerns to remain independent of the image concerns, while allowing their concurrent execution.

Overlapping Communication with Computation via Modularization of Concerns. Our next example presents a simple application for planning a trip. Planning requires finding available flights on the departure and return dates as well as a hotel and rental car for the duration of the trip. To find each of these items the program must communicate with services provided by other providers and each computation can be run independently.

Figure 14. An Image and Threshold Computation in Pañini.

```java
1 event Changed( Image pic ) {
2   class Percentile h;
3   int p /* Percentile value */
4   Percentile( int percentile ) {
5       register( this );
6       h = new Hashtable();
7       this.p = percentile;
8   }
9   when Changed do compute;
10   void compute( Image pic ) {
11       /* threshold is the intensity value for which cumulative
12           sum of pixel intensities is closest to the percentile p. */
13       h.add( pic, threshold );
14   }
15 }
16 class GlasbeyThreshold h;
17 GlasbeyThreshold() {
18       register( this );
19       h = new Hashtable();
20   }
21   when Changed do compute;
22   void compute( Image pic ) {
23       /* threshold is the intensity value for which cumulative
24           sum of pixel intensities has the most dominant value. */
25       values.put( pic, threshold );
26   }
27 }
28 class Image {
29   List pixels;
30   Image set( Integer i, Integer v ) {
31       pixels.setAt( i, v );
32       announce Changed( this );
33   }
34 }
```

Figure 15. Accessing service providers in handlers.

In this example the context variable tripData is used to both provide the handlers with information and to give the handlers a place to store their results. For example, class CheckAirline extracts source and destination information from the trip data and stores the flight results by calling the method setFlight. Similarly, the class CheckFlight computes and stores the hotel results and CheckRentalCar computes and stores the car rental search results. In this example as well Pañini’s design shows the potential of reconciling modularity goals with concurrency goals. When an event of type PlanTrip is announced each of the three handler methods can execute concurrently.

Performance Results. Modularization of the effects analysis and image analysis resulted in speedup of roughly 2x, whereas modularization of service requests gave speedups around 3x. These values were as expected based on the available concurrency in the problems. Moreover, this scalability is obtained without requiring programmers to write a single line of explicitly concurrent code.
6. Related Work

Events have a long history in both the software design and distributed systems communities. Pálini’s notion of asynchronous, typed events build on these notions, in particular recent work in programming languages focusing on event-driven design. In software design, events and implicit-invocation have been seen as a decoupling mechanism for modules, whereas in distributed systems, events are seen as a mechanism of decoupling component execution for location transparent deployment and extensibility.

A key difference between the programming models developed for event-based systems/message-passing systems/actor-based languages and that of Pálini is that the former assume that components in the system do not share state and only communicate by passing value types or record of value types, whereas the latter allows shared states (similar to mainstream languages like Java, C#) that is useful for many computation patterns. This means that if features from the former are adopted to mainstream languages as it is to decouple execution of components participating in an implicit-invocation design style, programmers will be directly responsible for ensuring that concurrent components do not have data races and deadlocks. Furthermore, reasoning about such systems will also be difficult due to concurrency.

In Pálini, programmers get concurrency benefits as a direct result of good design. Previous work on message-passing, publish/subscribe and actor-based languages either require programmers to manually account for data races, whereas Pálini automatically avoids all races.

Like Jade, Pálini is an implicit concurrency language. Programmers in Jade supply information about the effect of tasks so that the compiler may discover concurrency. Pálini is different in that it automates the process and removes the burden on the programmer to supply these effects by hand. Pálini also removes any errors which could be introduced by incorrect specification of effects. This is different from Grace which is an explicit threading language. Grace executes threads speculatively. If a conflict is detected, it rolls back the changes. Otherwise it commits the changes. Pálini detects conflict when handlers register.

Like X10, Pálini does not feature any construct for explicit locking. However, X10 is an explicit concurrency language and it uses atomic blocks for lock-free synchronization and uses the concept of atomic blocks as synchronization between activities. The Task Parallel Library (TPL) wraps computation into tasks and uses thread stealing as the underlying implementation. This is similar to Pálini’s runtime, but programmers in TPL have to explicitly account for races, whereas Pálini automatically avoids all races.

Similar to the effect sets of Pálini, deterministic parallel Java (DPJ/DPJizer) uses effect sets to provide deterministic semantics for programs. For DPJ/DPJizer, programmers explicitly write annotations on object fields, which ensures that fields are in separate regions. Then the tool infers summary for methods. Pálini does not require any specification. DPJ provides programmers with two concurrent constructs to parallelize their programs. This is unlike Pálini, which does not require programmers to construct explicitly parallel programs. Instead, Pálini promotes the goal of writing programs with good modular designs.

Pálini’s design is also not the first to promote implicit concurrency. For example, in POOL, Concurrent Smalltalk and BETA, objects implicitly execute in the context of a local process. This is different from Pálini where only handler instances are run implicitly and concurrently. This allows smoother integration with mainstream programming languages such as Java. This also permits an easier integration of our event-based model with the thread-based explicit concurrency models as promoted by Li and Zdancewic. In this work, we do not discuss the semantic issues with this integration, however.

Other recent work such as TaskJava and Tame have promoted similar integration with existing languages. For TaskJava, an asynchronous method is marked with async, indicating that it could block. This method may use a primitive wait to express its interests in a set of events and this expression will block until one of them fires. Similarly, Tame uses a primitive twait to block on events. In both these approaches, running of the concurrent task is explicitly managed by the programmer. In Pálini, however, handlers are implicitly spawned and managed by the language runtime. As a result, programmers are relieved of reasoning about locking and data race problems. Such software engineering properties are becoming very important with the increasing presence of concurrent software, increasing interleaving of threads in concurrent software, and increasing number of under-prepared software developers writing code using concurrency unsafe features.

Unlike Multilisp, which has the future construct, Pálini uses different expressions as synchronization points. Moreover, unlike Java’s current adoption of Futures, which is unsafe, heap access expressions in Pálini are safe. Furthermore, unlike previous work, Pálini doesn’t modify to the virtual machine.

7. Conclusion and Future Work

Language features that promote concurrency in program design have become important. Explicit concurrency features such as threads are hard to reason about and building correct software systems in their presence is difficult. There have been several proposals for concurrent language features, but none unifies program design for modularity with program design for concurrency. In the design of Pálini, we pursue this goal. In an effort to do so, we have developed the notion of asynchronous, typed events that are especially helpful for programs where modules are decoupled using implicit-invocation design style. Event announcements provide implicit concurrency in program designs when events are signaled and consumed. We have tried out several examples, where Pálini improves both program design and potential available concurrency. Unlike message-passing languages such as Erlang, the communication between implicitly concurrent handlers is not limited to value types or record of value types.

An important property of Pálini’s design is that, for systems utilizing implicit-invocation design style, it makes scalability a by-product of modularity. For example, observe that in genetic algorithm, AST analysis, image analysis, and trip planning addition of new modules in a non-conflicting manner doesn’t affect the scalability of existing modules. For example, a new observer for PlanTrip event (say sight seeing) would run concurrently with other observers. Similarly, a new thresholding observer could also run concurrently with other observers for Changed event.

Future work includes extending Pálini’s design, semantics and implementation in several dimensions. We have presented a conservative mechanism for detecting conflict between handlers, so it would be good to study and improve its precision. Furthermore, it would be sensible to investigate whether constructs similar to asynchronous, typed events can be developed for explicit invocation.

Acknowledgments

This work was supported in part by the US NSF under grant CCF-08-46059. Steven M. Kautz helped with concurrency analysis of examples.

References


