Isolating Determinism in Multi-threaded Programs

Lukasz Ziarek, Siddharth Tiwary, and Suresh Jagannathan

Purdue University
{lziarek, stiwary, suresh}@cs.purdue.edu

Abstract. Futures are a program abstraction that express a simple form of fork-join parallelism. The expression `future (e)` declares that `e` can be evaluated concurrently with the `future`'s continuation. *Safe*-futures provide additional deterministic guarantees, ensuring that all data dependencies found in the original (non-future annotated) version are respected. In this paper, we present a dynamic analysis for enforcing determinism of safe-futures in an ML-like language with dynamic thread creation and first-class references. Our analysis tracks the interaction between futures (and their continuations) with other explicitly defined threads of control, and enforces an *isolation* property that prevents the effects of a continuation from being witnessed by its future, indirectly through their interactions with other threads. Our analysis is defined via a lightweight capability-based dependence tracking mechanism that serves as a compact representation of an effect history. Implementation results support our premise that futures and threads can extract additional parallelism compared to traditional approaches for safe-futures.

1 Introduction

A *future* is a program construct used to introduce parallelism into sequential programs. The expression `future(`e`)` returns a future object F that evaluates e in a separate thread of control that executes concurrently with its continuation. The expression `touch(`F`)` blocks execution until F completes. A *safe-future* imposes additional constraints on the execution of a future and its continuation to preserve sequential semantics. By doing so, it provides a simple-to-understand mechanism that provides *deterministic parallelism*, transforming a sequential program into a safe concurrent one, without requiring any code restructuring. The definition of these constraints ensures that (a) the effects of a continuation are never witnessed by its future, and (b) a read of a reference r performed by a continuation is obligated to witness the last write to r made by the future.

In the absence of side-effects, a program decorated with safe-futures behaves identically to a program in which all such annotations are erased, assuming all future-encapsulated expressions terminate. In the presence of side-effects, however, unconstrained interleaving between a future and its continuation can lead to undesirable racy behavior. The conditions described above which prevent

such behavior can be implemented either statically through the insertion of synchronization barriers [1, 2], or dynamically by tracking dependencies [3], treating continuations as potentially speculative computations, aborting and rolling them back when a dependence violation is detected.

Earlier work on safe-futures considered their integration into sequential programs. In this context, the necessary dependence analysis has only to ensure that the effects of concurrent execution adhere to the original sequential semantics. As programs and libraries migrate to multicore environments, it is likely that computations from which we can extract deterministic parallelism are already part of multi-threaded computations, or interact with libraries and/or other program components that are themselves explicitly multi-threaded.

When safe-futures are integrated within an explicitly concurrent program (e.g., to extract additional concurrency from within a sequential *thread* of control), the necessary safety conditions are substantially more complex than those used to guarantee determinacy in the context of otherwise sequential code. This is because the interaction of explicitly concurrent threads among one another may indirectly induce behavior that violates a safe-future's safety guarantees. For example, a continuation of a future F created within the context of one thread may perform an effectful action that is witnessed by another thread whose resulting behavior affects F's execution, as depicted in the following program fragment:

```
let val x = ref 0
    val y = ref 0
in spawn( ...  future (... !y ...); x := 1);
   if !x = 1
   then y := 1
end
```

In the absence of the future annotation, the dereference of `y` (given as `!y`) would always yield `0`, regardless of the interaction between the future's continuation (here, the assignment `x := 1`) and the second thread[1]. (The expression `spawn`(e) creates a new thread of control to evaluate e with no deterministic guarantees.)

In this paper, we present a dynamic program analysis that tracks interactions between threads, futures, and their continuations that prevents the effects of continuations from being witnessed by their futures through cross-thread dataflow, as described in the example above. Our technique allows seamless integration of safe-futures into multi-threaded programs, and provides a mechanism that enables extraction of additional parallelism from multi-threaded code without requiring any further code restructuring, or programmer-specified synchronization. To the best of our knowledge, ours is the first analysis to provide lightweight thread-aware dynamic dependence tracking for effect isolation in the context of a language equipped with dynamic thread creation, first-class references, to support deterministic parallelism.

Our contributions are as follows:

[1] For the purposes of the example, we assume no compiler optimizations that reorders statements.

1. We present a dynamic analysis that *isolates* the effects of a continuation C from its future F even in the presence of multiple explicit threads of control. The isolation property is selective, allowing C to interact freely with other threads provided that the effects of such interactions do not leak back to F.
2. We introduce *future capabilities*, a new dependence analysis structure, that enables lightweight dynamic tracking of effects, suitable for identifying dependence violations between futures and their continuations.
3. We describe an implementation of futures and threads in MultiMLton, an optimizing compiler for Standard ML, and show benefits compared to traditional safe-futures.

2 Safe Futures and Threads

Safe futures are intended to ensure deterministic parallelism; they do so by guaranteeing the following two safety properties: (1) a future will never witness its continuation's effects and (2) a continuation will never witness the future's intermediate effects. In a sequential setting, the second condition implies a continuation *must* witness the *logically last* effects of a future.

To illustrate these properties, consider the two example programs given in Fig. 1. The code that is executed by the future is highlighted in gray. When a future is created it immediately returns a placeholder, which when touched will produce the return value of the future. A touch defines a synchronization point between a future and its continuation; execution following the touch are guaranteed that the future has completed.

Initially: x=0, y=0

Program 1

```
let fun f() = x := 1
    val tag = future (f)
in !x ; touch (tag)
end
```

Program 2

```
let fun f() = !y
    val tag = future (f)
in y := 1 ; touch (tag)
end
```

Fig. 1. Two programs depicting the safety properties that must be enforced to achieve deterministic parallelism for sequential programs annotated with futures.

In the program on the left, the future writes 1 to the shared variable `x`. The continuation (the code following `in`) reads from `x`. To ensure that the behavior of this program is consistent with a sequential execution, the continuation is only allowed to read 1. In the program on the right, the future reads from the shared variable `y` while the continuation writes to `y`. The future cannot witness any effects from the continuation, and therefore can only read 0 for `y`. To correctly execute the futures in programs 1 and 2 concurrently with their continuations

we must ensure that the future is isolated from the effects of the continuation and that the future's final effects are propagated to the continuation.

2.1 Interaction with Threads

The creation of new threads by either the future or the continuation requires reasoning about the created threads' actions. If a future internally creates a thread of control T, T's shared reads and writes *may* need to be isolated from the continuation's effects. We need to ensure that if a future witnesses a write performed by the thread it created, the future's continuation cannot witness any prior writes. On the other hand, a future *must* be isolated from the effects of any threads that a continuation creates. These threads are created *logically after* the future completes and therefore the future cannot witness any of their effects. To illustrate, consider the two examples programs given in Fig. 2.

Initially: x=0, y=0, z=0

Program 1

```
let fun g() = x := 1
    fun f() = spawn (g)
              if !x = 1
              then y := 2
    val tag = future (f)
in !x ; !y; touch (tag)
end
```

Program 2

```
let fun h() = z := 2
    fun f() = !z
    val tag = future (f)
in z := 1; spawn(h);
   touch (tag)
end
```

Fig. 2. Two programs depicting the safety properties that must be enforced to achieve deterministic parallelism for futures and continuations which spawn threads.

In the program on the left, the future creates a thread which writes to the shared variable `x`. The future then branches on the contents of `x`, and if `x` contains the value 1 it writes 2 to `y`. The continuation reads from both `x` and `y`. There are two valid outcomes: (1) if the body of the future executes before the assignment of `x` to 1 by the internally created thread, the continuation could read 0 for `x` and 0 for `y`; or (2), if the body of the future executes after the assignment by the internally created thread, the continuation would read 1 for `x` and 2 for `y`. An invalid result would be for the continuation to read `0` for `x` and 2 for `y` - this would imply that the continuation witnessed an intermediate value for `x` (here 0) that was not the last value witnessed by its future (which observed `x` to be 1). In the program on the right, the continuation creates a thread which writes to the shared variable `z`. Since the thread that the continuation creates should logically execute *after* the completion of the future, the future should only see the value 0 for `z`.

2.2 Transitive Effects

In the presence of multiple threads of control a future may incorrectly witness a continuation's effects transitively. Consider the sample program given in Fig. 3 that consists of two threads of control and one future denoted by the gray box. Notice that the future and its continuation do not access the same memory locations. The future simply reads from `y` and the continuation writes to `x`. Can the future read the value 2 from the shared variable `y`? Reading the value 2 for `y` would be erroneous because Thread 2 writes the value 2 to `y` only if `x` contains the value 1, implying the continuation's effects were visible to its future.

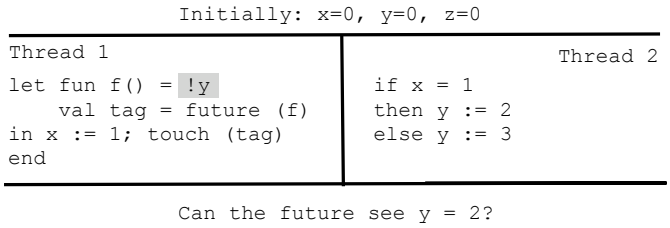

Fig. 3. Although the future and continuation do not conflict in the variables they access, the future, by witnessing the effects of Thread 2, transitively witnesses the effects of the continuation.

Similarly, a continuation may incorrectly witness a future's intermediate effects transitively (see Fig. 4). Here, functions `g`, `h`, and `i` *force* a particular interleaving between the future and Thread 2. It is incorrect for the continuation to witness the write of 1 to `z` by Thread 2 because Thread 2 subsequently overwrites `z` and synchronizes with the future by writing to `y` and then waiting until the future writes 2 to `x`. Thus, the continuation should only witness the value 2 for `z`.

As another example, consider the program given in Fig. 5 consisting of two explicitly created threads and one future denoted by the gray box. The functions `g()` and `h()` encode simple barriers that ensure synchronization between the future and Thread 2 - the future computation completes only after Thread 2 finishes. Is it possible for the continuation to read the value 1 for the shared variable `z`? Notice that the future does perform a write of 1 to `z` and in fact this is the last write the future performs to that shared variable. However, Thread 2 assigns 2 to `z` prior to assigning 2 to `y`. The future in Thread 1 *waits*, by executing the function `g`, until Thread 2 writes 2 to `y`. Therefore, the continuation *must* witness the write of 2 to `z` as this shared update logically occurs *prior* to the completion of the future. The future's write of 1 to `z` is guaranteed to occur prior to the write of 2 to `z` in Thread 2, since Thread 2 waits until the write to `x` to perform its update to `z`.

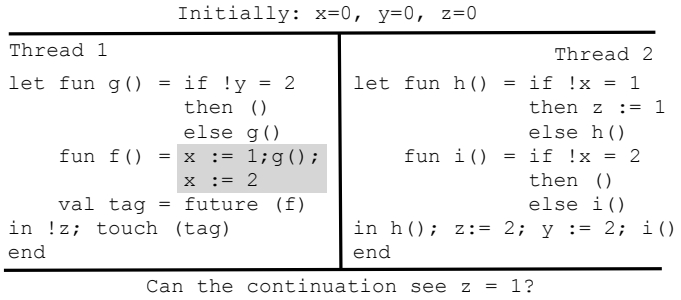

Fig. 4. Although the future and continuation do not conflict in the variables they access, the continuation may incorrectly witness the future's intermediate effects through updates performed by Thread 2.

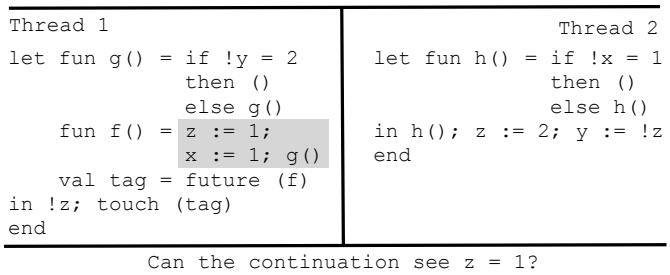

Fig. 5. The continuation cannot witness the future's write to `z` as this is an intermediate effect. The write to `z` in Thread 2 is transitively made visible to the continuation since the future synchronizes with Thread 2.

2.3 Future and future interaction

Similar issues arise when two futures witness each other's effects (see Fig. 6). Here, each thread creates a future; there are no dependencies between the future and its continuation. Assuming no compiler reorderings, the continuation of the future created by Thread 1 writes to `x` and the continuation of the future created by Thread 2 writes to `y`. The futures created by Thread 1 and 2 read from `y` and `x` respectively. It should be impossible for Thread 1 to read 1 from `y` and Thread 2 to read 1 from `x`. However, executing the futures arbitrarily allows for such an ordering to occur if the continuation of the future created by Thread 2 executes prior to the future in Thread 1 and the continuation of that future executes prior to the future created by Thread 2. In such an execution the futures witness values that logically should occur *after* their executions.

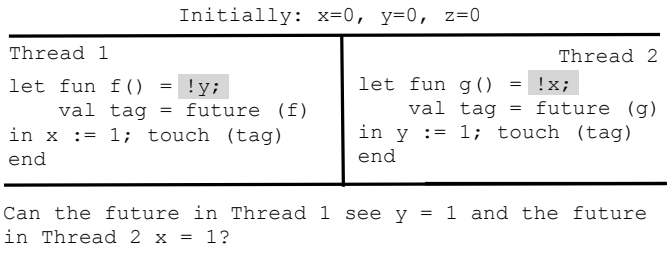

Fig. 6. Two futures created by separate threads of control may interact in a way that violates sequential consistency even though each future has no violations.

3 High Level Semantics

In this section, we define an operational semantics to formalize the intuition highlighted by the examples presented above; the semantics is given in terms of a core call-by-value functional language with first-class threads and references. The safety conditions are defined with respect to traces ($\bar{\tau}$). The language includes primitives for creating threads (`spawn`), creating shared references (`ref`), reading from a shared reference (!), and assigning to a shared reference (:=). We extend this core language with primitives to construct futures (`future`) and to wait on their completion (`touch`) (see Fig. 7).

Our language omits locking primitives; locks ensure that multiple updates to shared locations occur without interleavings from other threads. For our purposes, it is sufficient to track reads and updates to shared locations to characterize the safety conditions underlying the examples given in the previous section. As such, locks do not add any interesting semantic issues and are omitted for brevity.

In our syntax, v ranges over values, l over locations, e over expressions, x over variables, ℓ_f over future identifiers, ℓ_c to label computations associated with the continuation of a future, and `t` over thread identifiers. A program state is defined as a store (σ), a set of threads ($\overline{T}$), and a trace ($\bar{\tau}$). We decorate thread identifiers that are associated with futures with the identifier of the future, its continuation, or ϕ if the thread was created via a `spawn` operation. We assume future identifiers embed sufficient information about ancestry (i.e. futures created by another future or continuation) so that we can create fresh identifiers based on the parent's identifier ($fresh^I$) [2].

A trace ($\bar{\tau}$) is a sequence of actions represented by four types of trace elements: (1) $R(id, l)$ to capture the identifier id of the thread or future performing the read as well as the location (l) being read, (2), $W(id, l)$ defined similarly for writes, $S(id, id')$ to record spawn actions for a newly created thread with identifier id' created by thread id, and (4) $F(id, id')$ to record the parent/child relation for newly created futures either from other futures or threads.

$$
\begin{array}{rl}
v \in \mathit{Value} & \\
\ell_f, \ell_c \in \mathit{Id} & \\
\mathtt{t} \in \mathit{TID} & \\
l \in \mathit{Location} & \\
T \in \mathit{Thread} & := (\mathtt{t}^{Id}, e)
\end{array}
\qquad
\begin{array}{rl}
\mathit{id} \in \mathit{ID} & := \mathtt{t}^{\phi} + \mathtt{t}^{\ell_f} + \mathtt{t}^{\ell_c} \\
\sigma \in \mathit{Store} & := \mathit{Location} \xrightarrow{\text{fin}} \mathit{Value} \\
\tau \in \mathit{TraceElement} & := R(id, l) \ + \ W(id, l) + \\
 & \quad\ S(id, id) \ + \ F(id, id) \ + \ A(id, \_) \\
\overline{T} & := T \mid T \mid\mid \overline{T}
\end{array}
$$

$$
\begin{array}{rl}
e & := \mathtt{unit} \mid x \mid v \mid \lambda x.e \mid e\ e \\
 & \mid \ \mathtt{spawn}\ e \mid \mathtt{ref}\ e \mid \mathtt{touch}\ e \\
 & \mid \ e := e \mid !e \mid \mathtt{future}\ e
\end{array}
\qquad
\begin{array}{rl}
v & := \mathtt{unit} \mid l \mid \ell \mid \lambda x.e \\
E & := \cdot \mid E\ e \mid v\ E \mid \mathtt{touch}\ E \\
 & \mid \ E := e \mid l := E \mid \mathtt{ref}\ E \mid !E
\end{array}
$$

Safety

$$
\frac{\bar{\tau} \rightsquigarrow \bar{\tau'} \quad \mathit{safe}(\bar{\tau'})}{\mathit{safe}(\bar{\tau})}
\qquad
\frac{\forall \mathtt{t}^{\ell_f} \in \bar{\tau} \mid \mathit{max}(\bar{\tau}, \mathtt{t}^{\ell_f}) <_{\bar{\tau}} \mathit{min}(\bar{\tau}, \mathtt{t}^{\ell_c})}{\mathit{safe}(\bar{\tau})}
$$

Dependency Preserving Permutation

$$
\frac{\begin{array}{c}\tau_3 = A(id, \_) \quad A(id, \_) \notin \bar{\tau_2} \\ \bar{\tau} = \bar{\tau_1} : \bar{\tau_2} : \tau_3 : \bar{\tau_4} \\ \bar{\tau'} = \bar{\tau_1} : \tau_3 : \bar{\tau_2} : \bar{\tau_4} \\ \mathit{dep}(\bar{\tau}, \bar{\tau'})\end{array}}{\bar{\tau} \rightsquigarrow \bar{\tau'}}
$$

Inter-Thread Dependencies

$$
\frac{\begin{array}{c}\bar{\tau} = \bar{\tau_1} : \bar{\tau_2} : R(id, l) : \bar{\tau_3} \quad \bar{\tau'} = \bar{\tau_1} : R(id, l) : \bar{\tau_2} : \bar{\tau_3} \\ W(id', l) \notin \bar{\tau_2} \quad S(id', id) \notin \bar{\tau_2} \quad F(id', id) \notin \bar{\tau_2} \quad id' \neq id\end{array}}{\mathit{dep}(\bar{\tau}, \bar{\tau'})}
$$

$$
\frac{\begin{array}{c}\bar{\tau} = \bar{\tau_1} : \bar{\tau_2} : W(id, l) : \bar{\tau_3} \quad \bar{\tau'} = \bar{\tau_1} : W(id, l) : \bar{\tau_2} : \bar{\tau_3} \\ W(id', l) \notin \bar{\tau 2} \quad R(id', l) \notin \bar{\tau_2} \quad S(id', id) \notin \bar{\tau_2} \quad F(id', id) \notin \bar{\tau_2} \quad id' \neq id\end{array}}{\mathit{dep}(\bar{\tau}, \bar{\tau'})}
$$

Fig. 7. Language Syntax and Grammar

There are two safety rules that capture the notion of a well-behaved execution defined in terms of traces. The first rule states that an execution is safe if its trace enforces serial execution between all futures and their continuations. Serializability holds if the last action of a future precedes the first action of its continuation. The auxiliary relations *min* and *max* are defined in the obvious manner and return the first trace element and last trace element for a given identifier respectively. We use the notation $<_{\bar{\tau}}$ to order two trace elements in the trace $\bar{\tau}$.

The second rule defines an equivalence relation over safe traces in terms of a *dependency preserving permutation*: given an execution having a safe trace, any safe permutation of that trace (as defined by this relation) is also safe, and thus

App

$$\sigma, \bar{\tau}, (\mathtt{t}^I, E[(\lambda x.e)\ v]) \mid\mid \overline{T} \ \rightarrow\ \sigma, \bar{\tau}, (\mathtt{t}^I, E[e[v/x]]) \mid\mid \overline{T}$$

Ref

$$\frac{l \ \ \mathit{fresh} \quad \tau = W(\mathtt{t}^I, l)}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[\mathtt{ref}\ v]) \mid\mid \overline{T} \ \rightarrow\ \sigma[l \mapsto v], \bar{\tau}.\tau, (\mathtt{t}^I, E[l]) \mid\mid \overline{T}}$$

Touch

$$\frac{\overline{T} = (\mathtt{t}^{\ell_f}, v) \mid\mid \overline{T}'}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[\mathtt{touch}\ \ell_f]) \mid\mid \overline{T} \ \rightarrow\ \sigma, \bar{\tau}, (\mathtt{t}^I, E[v]) \mid\mid \overline{T}}$$

Spawn

$$\frac{\mathtt{t}' \ \ \mathit{fresh} \quad \tau = S(\mathtt{t}^I, \mathtt{t}'^{\phi})}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[\mathtt{spawn}\ e]) \mid\mid \overline{T} \ \rightarrow\ \sigma, \bar{\tau}.\tau, (\mathtt{t}'^{\phi}, e) \mid\mid (\mathtt{t}^I, E[\mathtt{unit}]) \mid\mid \overline{T}}$$

Future

$$\frac{\mathtt{t}' \ \ \mathit{fresh} \quad \ell_f, \ell_c \ \ \mathit{fresh}^I \quad \tau = F(\mathtt{t}^I, \mathtt{t}^{\ell_f})}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[\mathtt{future}\ e]) \mid\mid \overline{T} \ \rightarrow\ \sigma, \bar{\tau}.\tau, (\mathtt{t}^{\ell_c}, e) \mid\mid (\mathtt{t}'^{\ell_f}, E[\ell]) \mid\mid \overline{T}}$$

Read

$$\frac{\tau = R(\mathtt{t}^I, l)}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[!\ l]) \mid\mid \overline{T} \ \rightarrow\ \sigma, \bar{\tau}.\tau(\mathtt{t}^I, E[\sigma(l)]) \mid\mid \overline{T}}$$

Write

$$\frac{\tau = W(\mathtt{t}^I, l)}{\sigma, \bar{\tau}, (\mathtt{t}^I, E[l := v]) \mid\mid \overline{T} \ \rightarrow\ \sigma[l \mapsto v], \bar{\tau}.\tau(\mathtt{t}^I, E[\mathtt{unit}]) \mid\mid \overline{T}}$$

Fig. 8. Evaluation rules.

any execution that yields such a trace is well-behaved. The permutation rules preserve two types of dependencies: (1) intra-thread dependencies that ensure logical consistency of a given thread of control and (2) inter-thread dependencies that define a *happens-before* relationship among threads. The wild card trace element ($A(id, \_)$) matches any action performed by a future, continuation, or thread with the identifier id. A trace element can be permuted to the left of a series of actions $\bar{\tau_2}$ as long as that sub-trace does not contain any trace elements with the same identifier.

Inter-thread dependencies are defined by the relation *dep* that compares the permuted trace to the original trace. There are two rules that assert that inter-thread dependencies are preserved, one for reads and one for writes. The two relations mirror one another. A trace element $R(id, l)$ commutes to the left of a trace subsequence $\bar{\tau_2}$ if $\bar{\tau_2}$ does not contain an action performed by another that either writes to l (which would result in a read-after-write dependence), or does not spawn a thread or a future with identifier id. A similar right-mover [4] construction applies to writes.

The evaluation rules used to generate traces are given in Fig. 8, and are standard. To illustrate the rules, consider the unsafe execution of the program shown Fig. 6. Let the trace be $F(1, 3_f)$ $F(2, 4_f)$ $W(4_c, y)$ $W(3_c, x)$ $R(3_f, y)$ $R(4_c, x)$. Here, $F(1, 3_f)$ denotes the creation of the future by thread 1 with label 3_f, $F(2, 4_f)$ denotes the creation of the future in thread 2 with label 4_f, $W(4_f, \texttt{y})$ denotes the write of variable `y` by this future's continuation, $W(3_c, \texttt{x})$ denotes the write of variable `x` by future $\texttt{3}_\texttt{f}$'s continuation, $R(3_f, y)$ denotes the read of `y` by the first future, and $R(4_c, x)$ captures the read of `x` by the second future.

In the above trace, not all continuation actions occur after their future's. Because it is not a trivially serial trace, we need to consider whether it can be safely permuted. We can permute this trace to $F(1, 3_f)$ $F(2, 4_f)$ $W(4_c, y)$ $R(3_f, y)$ $W(3_c, x)$ $R(4_f, x)$; such a permutation preserves all inter-thread dependencies found in the original. But, no further permutations are possible; in particular, commuting $R(4_f, x)$ to the left of $W(4_c y)$ would break the dependency between $W(3_c, x)$ and $R(4_f, x)$. Similar reasoning applies if we permuted the actions of the second future with its continuation. Hence, we conclude the trace is unsafe.

4 Implementation

To enable scalable construction of safe futures, we formulate a strategy that associates *capabilities* with threads, futures, as well as the locations they modify and read. Abstractly, capabilities are used to indicate which effects have been witnessed by a future and its continuation, either directly or indirectly, as well as to constrain which locations a future and its continuation may read or modify. Capabilities ensure that *happens-before* dependencies are not established that would violate sequential consistency for a given thread of control. Thus, capabilities guarantee that an execution is equivalent to an execution where the future completes prior to its continuation's start in much the same way that the depedency preserving permutation asserts equivalence between traces.

A capability is defined as a binding between a label ℓ, denoting the dynamic instances of a future, and a *tag*. There are three tags of interest: `F` to denote that a thread or location has been influenced by a future, `C` to denote hat a thread or location has been influenced by a continuation, and `FC` to denote that a thread or location that first was influenced by a future and later by its continuation. It is illegal for a computation to witness the effects of a continuation and then the continuation's future (i.e., there is no `CF` tag). Tracking multiple labels allows us to differentiate between effects of different futures.

When a future with label ℓ is created, a constraint is established that relates the execution of the thread executing this future with its continuation. Namely, we add a mapping from ℓ to `F` for the thread executing the future and a mapping from ℓ to `C` for the thread executing the continuation. When the future or continuation reads or writes from a given location, we propagate its constraints to that location. Therefore, capabilities provide a *tainting* property that succinctly

records the history of actions performed by a thread, and which threads as well as locations those actions have influenced.

To ensure a thread T's read or write is correct, it must be the case that either (a) T has thus far only read values written by the future ℓ; (b) T has thus far only witnessed values written by the continuation of future ℓ; or (c) T had previously read values written by the future, but now only reads values written by the future's continuation. If T has previously read values written by the future ℓ, and then subsequently read values written by its continuation; allowing T to read further values written by the future would break correctness guarantees on the future's execution. Thus, prior to a read or a write to a location, if that location has capabilities associated with it, we must ensure that the thread which is trying to read or write from that location also has the same capabilities.

4.1 Capability Lifting

Capabilities can be lifted in the obvious manner. A capability can be lifted to a new capability that is more constrained. A thread or location with no capability for a given label ℓ can lift to either `C` or `F`. A thread which has a capability of `F` can lift the capability to `FC` and similarly a thread with a capability `C` can lift the capability to `FC`. A future and its continuation can never lift their capabilities for their own label. We allow a given thread to read or write to a location if its capabilities are equal to those for the given location. When a future completes, it is safe to discard all capabilities related to the future.

Based on capability mappings, we can distinguish between *speculative* and non-speculative computations. A continuation of a future is a speculative computation until the future completes. Similarly, any thread which communicates with a continuation of a future, becomes speculative at the communication point. On the other hand, any thread which has only `F` capabilities or an empty capability map is *non-speculative*. A future may in turn be speculative. As an example, consider the following program fragment which creates nested futures:

```
let fun g() = ...
    fun h() = ... future(g) ...
    fun i() = ... future(h) ...
in i()
end
```

At the point of the creation of the future to evaluate `g`, the remainder of the function `h` (the future evaluating `g`'s continuation) is speculative. The thread evaluating `g` as a future would have capabilities $\ell_g \mapsto$ `F` and $\ell_h \mapsto$ `F` and the thread evaluating the continuation would have capabilities $\ell_g \mapsto$ `C` and $\ell_h \mapsto$ `F`.

Using our notion of capabilities, we can handle future to future interactions by ensuring that the future and its continuation have consistent capabilities upon the futures completion. Since multiple threads of control can create futures (as in our example in Fig. 6) it is possible for a continuation of a future f to witness the effects of some other future g while the future f witnesses the effects of g's continuation. This would violate the dependencies imposed by sequential evaluation of both threads of control. To account for this, we check that a future

and its continuation have consistent capabilities when a future completes. In addition, when a location or thread that has a *speculative* capability (i.e. `C`, `FC`) acquires a capability for a future ℓ (i.e. $\ell \mapsto \texttt{F}$) we impose an *ordering* constraint between the future of the speculative capability and the future ℓ. Namely, the future ℓ logically *must* occur *after* the future of the continuation whose effect was witnessed. The manifestation of these checks directly mirrors the dependency preserving permutation rules described earlier.

4.2 Evaluation

To illustrate the benefits of our capability mechanism to provide safety in presence of *both* futures and threads, we tested our prototype implementation, comparing traditional safe-futures to threads and futures. All experiments were executed on a 16-way 3.2 Ghz AMD Opteron with 32 GB main memory. We executed the benchmark in two configurations. The first was a traditional safe-future implementation that leveraged capabilities for commit checks. This configuration did not include any mechanisms to track future to thread dependencies nor the rollback mechanism to revert multiple threads. The second configuration was our futures and threads implementation described above.

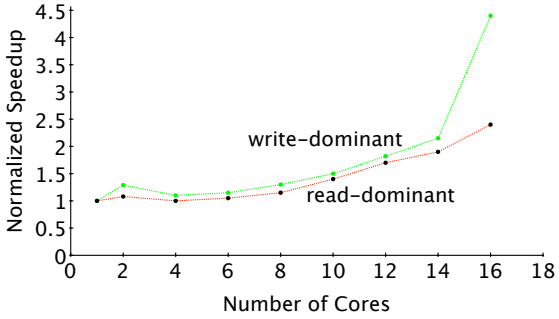

Fig. 9. Comparison of safe futures to futures and threads on all-pairs shortest path.

We tested both implementations on an all-pairs shortest path algorithm (Floyd-Warshall). The algorithm operates over a matrix representation of a graph of 16 million nodes. It executes a phase to calculate the shortest path through a given node in the graph, making the total number of phases proportional to the number of nodes in the graph. The algorithm takes as input the edge-weight matrix of a weighted directed graph and returns the matrix of shortest-length paths between all pairs of nodes in the graph. The algorithm works by first computing the shortest path between two given nodes of length one. It then computes the shortest path between the two nodes by increasing length.

Safe futures are able to extract parallelism between separate phases, allowing the computation of distinct phases in parallel. Although each phase depends on the phase prior, it is possible to execute multiple phases in parallel by staggering their executions. The amount of parallelism is limited by the dependencies between phases. The futures and threads implementation can not only extract parallelism between separate phases of the algorithm, but also parallelism within a phase. This is accomplished using fork-join parallelism, and is not easily expressible using only safe futures without significant modifications to the program structure. We observe that threads can be allocated to compute over different parts of the matrix within a given phase by splitting the matrix into conceptual chunks. Although this is easily expressed using threads, safe-futures require the programmer to express the splitting of the matrix via control flow within each phase. This occurs because there does not exists a mechanism to constrain the execution of a future over a part of a data structure.

In addition, a safe futures-only strategy would enforce serializability between each chunk, even though this is not necessary. Results are summarized in Fig. 9. Notice that after executing 6 phases in parallel, the benefits of using only safe futures decreases. This occurs due to the higher rate of aborts from the dependencies between phases. In this workload, one future is created for each available processor meaning that on four cores four phases are executed in parallel at any given moment. In all cases roughly 16 million futures are created (one per phase), but the number of phases executed in parallel depends on the availability of processors. In the futures and threads implementation, we create one thread for every two available processors per each phase. Each thread is split into a future and continuation, allowing the computation of two phases in parallel. The total number of threads (and futures) is therefore 16 million times the number of cores/2. In both implementations, it is not beneficial to create more speculative work than the amount of processors available.

Fig. 9 shows the benefits of using futures with threads on two different types of workloads for the all-pair shortest path problem, one containing 5% writes (read-dominant) and the other 75% writes (write-dominant). The workloads are generated using the observation that paths through nodes which are ranked higher in the matrix are utilized in later phases. If the weights of edges between higher ranked nodes are smaller more writes occur since new smaller paths are found in successive phases. The read-dominant workload, on the other hand, is generated by making the edge-wights between lower ranked nodes smaller than the those between high ranked nodes. We see that futures with threads outperform using only safe futures in both styles of workload. In the read-dominant workload, the number of aborts for having futures-only is roughly 2 times higher (5432 aborts total). In comparison, in the write-dominant workload, the number of aborts for having just safe futures goes as high as 5 times (around 25255 aborts total) more than having both futures and threads. In both the workloads, we see more aborts due to the large number of cross phase data dependencies. This is especially true when speculating across more than four phases of the algorithm as the benefits of staggering executions becomes muted. The write-

dominant workload results in more aborts as the number of data dependencies between any two phases increases. The above experiment illustrates the benefits of using futures with threads over just safe futures for a benchmark which is more write-dominant. This occurs because the futures with threads scheme can extract parallelism from within each phase, thereby limiting the number of parallel speculative phases necessary to keep all processors busy.

5 Related Work and Conclusion

Futures are a well known programming construct found in languages from Multilisp [5] to Java [6]. Many recent language proposals [7–9] incorporate future-like constructs. Futures typically require that they are manually claimed by explicitly calling `touch`. Pratikakis et. al [10] simplify programming with futures by providing an analysis to track the flow of future through a program and automating the injection of claim operations on the future at points where the value yielded by the future is required, although the burden of ensuring safety still rests with the programmer.

There has been a number of recent proposals dealing with *safe-futures*. Welc et. al [3] provide a dynamic analysis that enforces deterministic execution of sequential Java programs. In sequential programs, static analysis coupled with simple program transformations [1] can ensure deterministic parallelism by providing coordination between futures and their continuations in the presence of mutable state. Unfortunately neither approach provided safety in the presences of exceptions. This was remedied in [11, 2] which presented an implementation for exception-handling in the presence of *safe futures*.

Flanagan and Felleisen [12] presented a formal semantics for futures, but did not consider how to enforce safety (i.e. determinism) in the presence of mutable state. Navabi and Jagannathan [13] presented a formulation of safe-futures for a higher-order language with first-class exceptions and first-class references. Neither formulation consider the interaction of futures with explicit threads of control. Futures have been extend with support for asynchronous method calls and active objects [14]. Although not described in the context of safe-futures, [15] proposed a type and effect system that simplifies parallel programming by enforcing deterministic semantics. Grace [16] is a highly scalable runtime system that eliminates concurrency errors for programs with fork-join parallelism by enforcing a sequential commit protocol on threads which run as processes. Boudol and Petri [17] provide a definition for valid speculative computations independent of any implementation technique. Velodrome [18] is a sound and complete atomicity checker for multi-threaded programs that analyzes traces of programs for atomicity violations.

This paper presents a dynamic analysis for enforcing determinism in an explicitly concurrent program for a higher-order language with references. Safety is ensured dynamically through the use of a light weight capability tracking mechanism. Our initial prototype indicates that futures and threads are able to extract additional parallelism over a traditional safe-future approach.

Acknowledgements: This work is supported by the National Science Foundation under grants CCF-0701832 and CCF-0811631.

References

1. Navabi, A., Zhang, X., Jagannathan, S.: Quasi-static Scheduling for Safe Futures. In: PPoPP, ACM (2008) 23–32
2. Navabi, A., Zhang, X., Jagannathan, S.: Dependence Analysis for Safe Futures. Science of Computer Programming (2011)
3. Welc, A., Jagannathan, S., Hosking, A.: Safe Futures for Java. In: OOPSLA, ACM (2005) 439–435
4. Flanagan, C., Qadeer, S.: A Type and Effect System for Atomicity. In: PLDI. (2003) 338–349
5. Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation. ACM Trans. Program. Lang. Syst. **7** (1985) 501–538
6. http://java.sun.com/j2se/1.5.0/docs/guide/concurrency.
7. Allan, E., Chase, D., J. Hallett, V.L., Maessen, J., Ryu, S., Steele, G., Tobin-Hochstadt, S.: The Fortress Language Specification Version 1.0. Technical report, Sun Microsystems, Inc. (2008)
8. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, ACM (2005) 519–538
9. Liskov, B., Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In: PLDI, ACM (1988) 260–267
10. Pratikakis, P., Spacco, J., Hicks, M.: Transparent Proxies for Java Futures. In: OOPSLA, ACM (2004) 206–223
11. Zhang, L., Krintz, C., Nagpurkar, P.: Supporting Exception Handling for Futures in Java. In: PPPJ, ACM (2007) 175–184
12. Flanagan, C., Felleisen, M.: The semantics of future and an application. Journal of Functional Programming **9** (1999) 1–31
13. Navabi, A., Jagannathan, S.: Exceptionally safe futures. In: COORDINATION, Berlin, Heidelberg, Springer-Verlag (2009) 47–65
14. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: ESOP, Springer-Verlag (2007) 316–330
15. Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel java. In: OOPSLA. (2009)
16. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for c/c++. In: OOPSLA, New York, NY, USA, ACM (2009) 81–96
17. Boudol, G., Petri, G.: A Theory of Speculative Computation. In: ESOP. (2010) 165–184
18. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI. (2008) 293–303