Quasi-Static Scheduling for Safe Futures

Armand Navabi Xiangyu Zhang Suresh Jagannathan
Purdue University, Department of Computer Science, West Lafayette, Indiana 47907
{anavabi,xyzhang,suresh}@cs.purdue.edu

Abstract

Migrating sequential programs to effectively utilize next generation multicore architectures is a key challenge facing application developers and implementors. Languages like Java that support complex control- and dataflow abstractions confound classical automatic parallelization techniques. On the other hand, introducing multithreading and concurrency control explicitly into programs can impose a high conceptual burden on the programmer, and may entail a significant rewrite of the original program.

In this paper, we consider a new technique to address this issue. Our approach makes use of *futures*, a simple annotation that introduces asynchronous concurrency into Java programs, but provides no concurrency control. To ensure concurrent execution does not yield behavior inconsistent with sequential execution (i.e., execution yielded by erasing all futures), we present a new interprocedural summary-based dataflow analysis. The analysis inserts lightweight barriers that block and resume threads executing futures if a dependency violation may ensue. There are no constraints on how threads execute other than those imposed by these barriers.

Our experimental results indicate futures can be leveraged to transparently ensure safety and profitably exploit parallelism; in contrast to earlier efforts, our technique is completely portable, and requires no modifications to the underlying JVM.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Algorithms, Languages, Reliability, Performance

Keywords concurrency control, future, static program analysis

1. Introduction

Migrating existing sequential programs to next-generation multicore and many-core architectures is a key challenge confronting application developers, implementors, and architects. In languages such as Fortran in which computation is mostly expressed analyzable control-flow abstractions (e.g., loops), automatic parallelization techniques that expose concurrent execution across loop iterations is a feasible way to exploit available parallelism. In languages like Java which include more complex dataflow and control-flow mechanisms, these techniques are not likely to be as effective. While programmers can leverage Java's support for multi-threading to expose concurrency, this flexibility comes at a price. Rewriting programs to be explicitly multi-threaded is non-trivial, requiring deep knowledge of the program's intended behavior. Well-known problems such as data races, deadlocks, etc. can easily occur. Recent work that leverages higher-level abstractions such as software transactions [12, 14] can alleviate some of these issues, but the onus still remains on the programmer to inject transactions (and related concurrency abstractions) correctly, to ensure that the resulting program maintains the invariants assumed by the original.

In this paper, we consider an alternative that exploits both the analytical capability of compilers, and a programmer's domain-specific knowledge. Informally, parallelizing a program consists of two major tasks: (a) identifying program points where concurrent execution may be initiated, and (b) incorporating concurrency control to ensure threads access shared data safely, without violating intended dependencies. The former is arguably simpler than the latter: making an unwise choice for where concurrency should be introduced can lead to poor performance, but failure to correctly protect shared data can lead to erroneous results. Building upon this intuition, our approach only requires programmers to specify opportunities for concurrency, *without* requiring them to also specify concurrency control. Our goal is to provide a portable and lightweight, mostly transparent, translation mechanism for languages like Java that allows simply annotated sequential programs to exploit parallel computing resources when possible. While it is true that in some cases effective utilization of these resources will require a major restructuring of the original sequential program, including the introduction of possibly complex concurrency control mechanisms, we believe that there is a broad class of applications for which compiler analysis, along with minimal runtime support can be leveraged to avoid such drastic surgery.

We use *futures* as our concurrency abstraction. Futures are found in the `java.util.concurrent` package which is part of the Java 2 Platform Standard Edition 5.0. The future interface is extremely simple, and effectively serves as an annotation on method calls that allows the call to be executed asynchronously. A subsequent *claim* operation on the future serves as a barrier that blocks until the future completes. Simply using futures to introduce concurrency into Java programs is unfortunately insufficient. Without appropriate safeguards such as locks to ensure concurrently executing futures access shared data correctly, a future annotated program may exhibit races, deadlocks, and other ill-desired behavior. Notably, even the introduction of appropriate synchronization on shared data may be insufficient since interleaving execution among futures may lead to race and deadlock-free executions that are nonetheless inconsistent with the behavior of the original (future-erased) sequential program.

Earlier work on this subject has adapted techniques based on thread-level speculation and transactional memory to enforce desired safety properties. The basic idea is to execute threads representing futures in a conceptual sandbox. If a thread manipulates shared data in ways that violate program dependencies, it is re-

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

voked, and all of its effects discarded. By requiring the runtime to detect and remedy dependency violations, these approaches free the programmer from explicitly weaving a concurrency control protocol within the application. Unfortunately, these techniques also require hardware support [21, 16] or a heavyweight runtime [30]. Because sandbox maintenance requires support for object versioning, operation logging, additional metadata on object headers, etc., performance results can vary widely, and in some cases, *decrease* substantially. More importantly, portability is sacrificed since incorporating these mechanisms entails non-trivial low-level modifications to the JVM and/or underlying architecture.

In this paper, we explore techniques that shift much of the burden for enforcing safety from the runtime to the compiler. We present a novel summary-based interprocedural static analysis on programs annotated with futures, and a program transformation that injects synchronization primitives based on the analysis outcome. These primitives only require lightweight runtime support. The idea is that a synchronization barrier is introduced whenever a potential dependency violation as defined by the sequential semantics may occur. Notification barriers are also inserted whenever a potentially offending shared access operation completes. A read or write operation on shared data also manipulated by futures instantiated earlier must block until these futures complete their accesses. We describe both the compiler framework and the runtime mechanisms used to inject and execute these barriers. There are no other constraints on how threads execute other than those imposed by these barriers.

This paper makes the following contributions:

- We propose a technique called *quasi-static scheduling* that automatically inserts lightweight synchronization primitives to ensure safety given a future-annotated program. Our safety criteria ensures that execution of the program will have the same observable (deterministic) behavior as the sequential program derived by erasing all futures. Our analysis is interprocedural, and handles many Java features including dynamic threading.
- We devise a novel lightweight runtime that distributes concurrency control defined by the synchronization barriers inserted by the analysis among the threads executing the computation encapsulated by futures, rather than requiring the use of a centralized scheduler.
- We experimentally evaluate our techniques on a set of Java benchmarks including some taken from the Java Grande [27] and DaCapo [3] benchmark suites, OO7 [6], an object-oriented database application, and a red-black tree implementation [1]. Our results confirm our hypothesis that quasi-static scheduling can be an effective technique to easily migrate sequential programs to multicore systems with minimal restructuring.

2. Overview

Consider the example shown in Fig. 1(a). Method `foo(...)` performs various operations on object `x` depending on the parameter `op`. More specifically, it read-accesses `x` if `op==1` and write-accesses `x` if `op==2`. In this sample program, `foo` is called four times, performing in order two read operations, a write operation, and then another read operation on the same object `x`. Symbol $\texttt{foo}^i$ denotes the ith call of method `foo`.

Suppose we wish to execute the calls to `foo` in parallel. Concurrent execution of these calls must still adhere to the dependency requirements imposed by sequential evaluation: a read access performed in one call must not witness a write access performed by a later one, and a write access in one call must not follow a read performed by a later one. Thus, we wish to ensure that concurrent

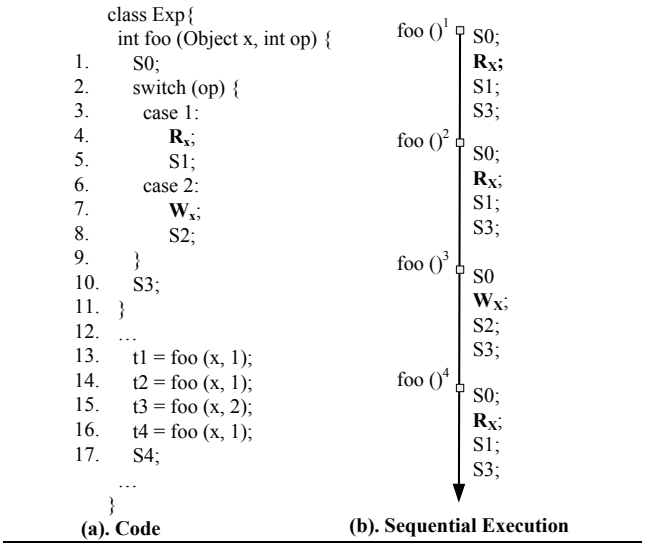

Figure 1. A Motivating Example.

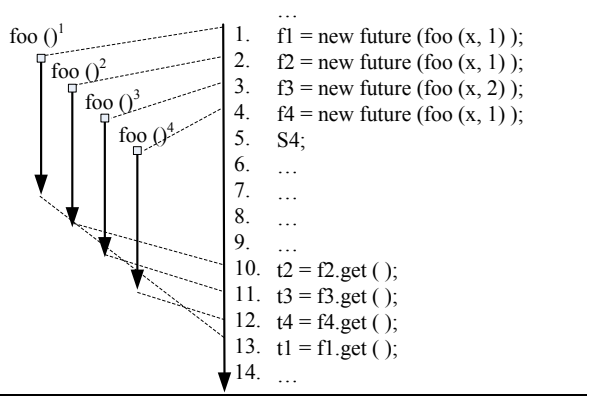

Figure 2. Concurrent Evaluation Using Futures.

evaluation of different calls to `foo` still yields deterministic behavior consistent with a sequential version of the same program.

To introduce concurrency, we use *futures*, a simple programming abstraction that permits the introduction of asynchronous computation into a sequential program. While first introduced in the context of Lisp, futures are also found in the `java.util.concurrent` package in the Java 2 Platform Standard Edition 5.0.

In Fig. 2, a call to `future()` divides the current execution into two concurrent parts, one being the method call passed as the parameter, e.g., `foo(..)`, which we refer to as the *future*, and the other the remainder of the computation, which we refer to as the future's *continuation*. For instance, the first call `future()` spawns a future `f1`, which executes the `foo` method; the remaining execution, which includes three other calls to `foo`, `S4`, etc., constitutes the continuation of `f1`. A future is claimed through the function `get()` when the result of the computation produced by the future is needed. At the claim point, the future and its continuation join. For example, `f1` is claimed at line 13 in Fig. 2. Note that a continuation or a future can further spawn futures and continuations, e.g., futures `f3` and `f4` are part of the continuation of `f2`.

Significantly, the translation from the sequential program in Fig. 1(a) to the concurrent version using futures does not introduce any concurrency control: there are no locks, transactions, or other synchronization mechanisms that enforce race-free accesses to shared data manipulated within `foo`. Moreover, while concurrency control mechanisms like locks can be used to guarantee race-

freedom, our requirements are stronger: we are only interested in concurrent executions in which the observable behavior of the program is consistent with a sequential execution of the same program with all future annotations erased (i.e., the execution in Fig. 1(b)). In this example, enforcing this condition requires that the write access performed by `f3`, which is part of the continuation of future `f2`, wait for `f1` and `f2` to finish their read accesses. Similarly, the read access performed by future `f4` must wait for the write access in `f3` to complete. Simply protecting concurrent accesses to shared data using locks cannot enforce these constraints.

Previously, researchers have proposed speculative thread spawning and lazy recovery facilitated by software transactional memory [30, 29] to address these safety issues. Fig. 3 explains the way safe futures [30] parallelize our sample program. The four `foo` operations are speculatively spawned as separate threads and run in parallel. We elide details about the spawn and get operations for brevity. In the safe futures approach, versioning is used to tolerate shared write accesses, i.e., a shared write to an object creates a new version of the object. A shared read searches for the right version. For example in Fig. 3, the write access by the third future generates a new version of `x`. Although the read in `foo()`^{2} happens after the write, it gets the correct value from an older version. If a read occurs before a write to the same variable which is in the read's logical past, e.g., the read in `foo()`^{4}, the continuation performing the offending read has to be revoked and re-executed. Such a method supports speculative parallelization through a heavyweight runtime without requiring static dependency analysis.

The use of a heavyweight runtime to track read and write accesses significantly limits the simplicity benefits of safe speculative asynchronous methods for a number of reasons: (1). Rollback and re-execution happen frequently if shared writes are frequent, and can substantially degrade performance; (2). Employing STM machinery to commit updates, detect conflicts, and version data for fine-grained futures can be expensive because maintenance of associated data is performed at every shared access; (3) The runtime has the typical limitations of most STMs such as the incapability to effectively handle irrevocable actions such as I/O.

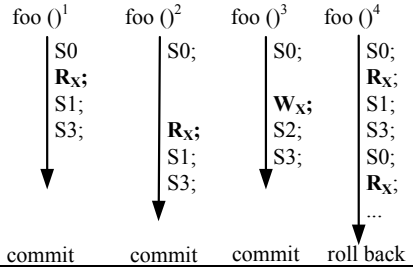

Figure 3. Transaction Runtime Support.

We propose a technique that supports speculative thread spawning and guarantees our desired safety property by compiler inserted synchronizations that only require lightweight runtime support. The basic idea is that *a shared access performed by a continuation can not be discharged until all other futures in its logical past have completed performing any conflicting accesses.* For example in Fig. 1(a), the read access at line 4 in a future can only be discharged if all other futures in its logical past have either (1) entered the `op==1` branch, implying the future will not subsequently perform any conflicting write access, or (2) finished the write access.

Fig. 4 illustrates how we might enforce these conditions. The read access at line 5 is preceded by the barrier ***allowed(a_x)***, which only completes when all futures that are in the logical past have granted condition $\boldsymbol{a_x}$, namely they have either entered a branch in which no further conflicting write access to x will be performed, or they have finished performing the write. Thus, a condition is a guard on shared data accesses. In this example, the guard dictates when read accesses to shared variable `x` can take place. The instrumentations at 3 and 9 grant the permission. The ***allowed(a_x)*** barrier requires a total logical ordering among futures (a future f' created within a continuation of a future f is logically ordered after f); we discuss efficient non-centralized approaches to enforce this ordering in Section 3.3.

Note that similar instrumentation is required for the write access. We omit it in Fig. 4 for brevity. Given this instrumentation, the resulting runtime schedule is shown in Fig. 5. The ***grant*** barrier in the first future discharges the ***allowed*** in the second future, which requires all logically preceding futures (the first future in this case) to grant permission. The ***allowed*** in the first future returns immediately as it has no preceding future. The ***allowed*** in the fourth future does not return until the three preceding futures grant the condition.

The placement of ***grant*** barriers is key to the degree of parallelism achieved. In our example, a future immediately issues a grant after it enters the `op==1` branch and discharges other futures so that the read access and its following statements `S1` and `S3` can run in parallel with other futures. A sub-optimal solution is to have the ***grant*** barriers at lines 3 and 9 combined and placed after the `switch` block. Despite being safe, it unnecessarily prevents $\mathtt{R}_x$; `S1` from running concurrently with statements in other futures or continuations. Note that if the ***allowed*** barrier is placed at the beginning of a future and ***grant*** barrier is placed at the end, the resulting scheduling induces a completely sequential execution.

```
    ...
1.  switch (op) {
2.    case 1:
3.      grant(ax);
4.      allowed(ax);
5.      Rx;
6.      S1;
7.    case 2:
8.      Wx;
9.      grant(ax);
10.     S2;
11.   ...
```

Figure 4. Instrumentation.

Compared to a STM-inspired solution [30, 29], our approach features a very lightweight runtime. Synchronization conditions such as the parameter $\boldsymbol{a_x}$ in Fig. 4 are assigned at every program point that performs a shared access instead of per object. Our technique requires no rollback support since the placement of ***allowed*** and ***grant*** barriers guarantees safety. While a static analysis leads to low runtime overheads, the approximations it uses to enforce dependencies may be more conservative than necessary. For example, if the write access in the execution in Fig. 1 is for a different object than that accessed by concurrent reads, but the analysis concludes they are potentially aliases, the opportunity to execute all operations in the four futures concurrently would be missed. Nonetheless, we believe our technique is better suited to facilitate easy migration of sequential programs to multi-core environments because it does not burden implementations with the significant requirement of a highly-tuned sophisticated runtime to enforce safety.

3. Analysis

From the example in the previous section, the challenges entailed by our techniques become clear. First, we need a static analysis that identifies all program points that perform conflicting accesses. Second, we need to insert synchronizations accordingly and ensure that they are deadlock free. Third, we need to devise a lightweight runtime to provide implementation support for these annotations.

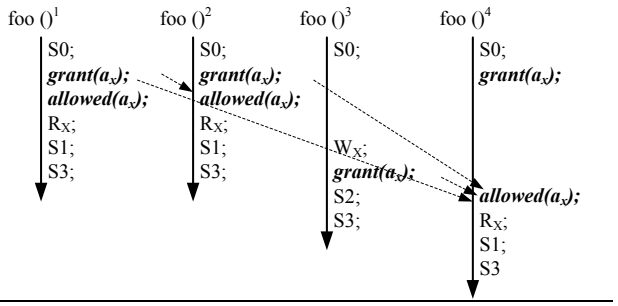

Figure 5. Runtime Schedule.

3.1 Future Analysis

A *future analysis* computes the set of futures that may run in parallel with a program point *l*. In other words, *l* belongs to the continuations of these futures. Based on the analysis result, ***allowed*** and ***grant*** barriers are inserted to enforce the safety property.

```
     void A ( )
1.   {
2.       x = new (…);
3.       B (x);
4.       if (x.future != NULL) {
5.           x.future.get( );
6.       }
7.   }

     void B (... o )
8.   {
9.       f1 = new future (…);
10.      if (… )
11.          C (f1);
12.      else
13.          o.future = f1;
14.  }
15.  void C (… y)
16.  {
17.      y.get ( );
     }
```

Figure 6. A more complex example illustrating the need for interprocedural analysis.

Ordinarily, the return point for a method or procedure call immediately follows the call-site, in the absence of exceptions. In a method call encapsulated within a future, this property need not hold. Indeed, it is likely that the point at which the future is claimed is far removed from the point at which future is created. As shown in Figure 6, a future is spawned in method `B`, and is claimed either in `C`, or escapes from `B` and is then claimed in `A`. Determining the set of active futures at any given point becomes even more complex in the presence of aliasing.

To analyze the set of futures that can run in parallel for a program point in circumstances as described above, we perform a *future analysis* on a program representation called a *Future Spawning Graph* (FSG). If a method is spawned as a future, a *spawn edge* is introduced between the spawning point in the caller and the callee. A *claim edge* is introduced between the exit of a spawned method and its corresponding claim point, indicated by a call of the *get()* method. Figure 7 shows part of the FSG for the example in Section 2. As can be seen, for futures `f1` and `f2`, two spawn edges `L1` → `L3`, `L2` → `L3`, and two claim edges `L4` → `L5`, `L4` → `L6` are inserted.

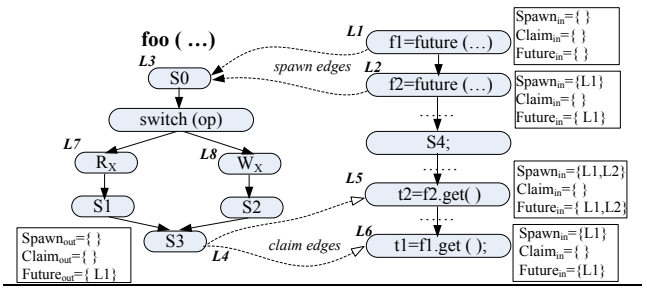

Figure 7. The Future Spawning Graph.

Future analysis is a summary-based interprocedural data flow analysis (see Fig. 8). Functions summaries are computed in a bottom-up phase and the final results are computed in a following top-down phase. Fig. 8 presents the detail of the analysis. It assumes the results of a may points-to analysis *PT*. In the bottom-up phase, given a method *m*, a function summary is computed as the transfer function. *Spawn* represents the set of futures that are spawned in *m* or other functions called by *m* and not yet claimed. Note that futures are identified by their static allocation sites. *Claim* represents the set of claims of futures spawned outside *m*, which is mainly used to construct the function summary. Both *Spawn* and *Claim* entail forward computation. *Spawn* requires a union operation at a join point, whereas *Claim* requires an intersection since a future can be considered claimed only if it is claimed along all incoming paths.

We consider the computation of the *Spawn* set for each statement in turn. In the case of a spawn statement `l: future(m(...))` (where `l` is the label denoting this program point), the $Spawn_{out}$ set is computed by discarding elements in the kill-set found in the summary of each method *T* that `m` may refer to, and adding the corresponding elements in *T*'s gen-set. Label `l` is also added to the $Spawn_{out}$ set as the future spawned at `l` may run concurrently with statements following the creation of the future. In the case of a function call `m(...)` that is not spawned as a future, $Spawn_{out}$ is computed similarly, except the label is not added.

Consider a program point *g* that spawns a future *f*, and suppose *f* is claimed in some method *m* different from the method in which *g* is found. The claim summary for *m* can include *g* provided that there is no cyclic path (interprocedural or otherwise) through *g* that does not include a claim on *f*. This condition is necessary to ensure that a claim on a future indeed kills all instances of that future. For example, in the program below, although $F[j]$ at 6 must point to $F[i]$ at 2, it cannot be easily determined statically whether the `get()` operation at line 6 claims all the futures spawned at line 2. Consequently, we have to assume that statement `S0` at 8 may run concurrently with any of the futures created at line 2.

```
1: for (i=...) {
2:       F[i]= new future (...);
3: }
4: . . .;
5: for (j=...) {
6:       F[j].get();
7: }
8: S0;
```

Computation of the claim set follows the same reasoning as above. In the presence of a call to method `m`, either spawned within a future or executed sequentially, the $Claim_{out}$ set at a program point is computed as the union of the $Claim_{in}$ at that point, and the intersection of the $SumClaim()$ sets over all methods that `m` may point to. In the presence of a call to `f.get()`, if `f` must point to a future spawned externally at *g* and *g* cannot reach itself without encountering its claim, *g* is added to $Claim_{out}$.

Finally, *SumSpawn* and *SumClaim* are just the corresponding *Spawn* and *Claim* sets at the exit point of the corresponding method.

During the second top-down phase, dataflow facts are propagated from callers to callees to compute the final *future* sets for each statement. We conservatively compute the $Future_{out}$ of a method `m`'s entry point as the union of the $Future_{in}$ sets of all call sites to *M*.

Example. Figure 7 shows an example of the results computed by the analysis. During the bottom-up phase, method `foo` is first analyzed. Since it does not spawn or claim any futures, it has the summary *SumSpawn=SumClaim=*ϕ. This result is used in the analysis of the main body. Thus, $Spawn_{in}(\boldsymbol{L2}) = Spawn_{out}(\boldsymbol{L1}) = \{\boldsymbol{L1}\}$

Assumptions: PT : `label` $\rightarrow \mathcal{P}$(`abstract location`) represents the result of a may points-to analysis. Abstract locations can be object allocation sites, methods or future spawn sites. $Single$ yields the points-to set of its argument if that set holds a single abstract location, and the empty set otherwise. F_M denotes intraprocedural control flow of method M, IF denotes interprocedural control flow.
Output: $Future$: `label` $\rightarrow \mathcal{P}$(`abstract location`) represents the set of futures that may run in parallel with a statement.
Initialization:
$\forall$ label l in method m, $Spawn_{out}(l) = \phi,\ Claim_{out}(l) =$ `Unknown`

The bottom-up phase:

$$Spawn_{in}(l) := \bigcup_{(p,l)\in F_{\mathtt{M}}} (Spawn_{out}(p))$$

$$Claim_{in}(l) := \bigcap_{(p,l)\in F_{\mathtt{M}}} (Claim_{out}(p))$$

$$Spawn_{out}(\mathtt{l}) := \begin{cases} (Spawn_{in}(l) - \bigcap_{T\in PT(\mathtt{m})} SumClaim(T)) \bigcup (\bigcup_{T\in PT(\mathtt{m})} SumSpawn(T)) \bigcup \{\mathtt{l}\} & l : \texttt{future (m(...));} \\ (Spawn_{in}(l) - \bigcap_{T\in PT(\mathtt{m})} SumClaim(T)) \bigcup (\bigcup_{T\in PT(\mathtt{m})} SumSpawn(T)) & l : \texttt{m(...);} \\ Spawn_{in}(l) - \{g \mid g \in Single(f) \text{ and there is not a path} \\ \quad P : g \rightarrow ... \rightarrow g \text{ s.t. } g \text{ is not claimed during } P\} & l : \texttt{f.get();} \\ Spawn_{in}(l) & otherwise. \end{cases}$$

$$Claim_{out}(l) := \begin{cases} Claim_{in}(l) \bigcup (\bigcap_{T\in PT(m)} SumClaim(T)) & l : \texttt{future (m(...))} \\ & or\ \texttt{l: m(...);} \\ Claim_{in}(l) \bigcup \{g \mid g \in Single(f) \text{ and } g \text{ is not local.} \\ \quad \text{and there is not a path } P : g \rightarrow ... \rightarrow g \text{ s.t. } g \text{ is not claimed during } P\} & l : \texttt{f.get();} \\ Claim_{in}(l) & otherwise. \end{cases}$$

$SumSpawn(\mathtt{m}) := Spawn_{out}(EXIT)$.
$SumClaim(\mathtt{m}) := Claim_{out}(EXIT)$.

The top-down phase:

$$\begin{aligned} Future_{out}(ENTRY_{main}) &:= \phi \\ Future_{out}(ENTRY_M) &:= \bigcup_{(p,ENTRY_M)\in IF} (Future_{in}(p)) \\ Future_{in}(l) &:= \bigcup_{(p,l)\in F_M} (Future_{out}(p)) \\ Future_{out}(l) &:= (Future_{in}(l) - Claim_{out}(l)) \bigcup Spawn_{in}(l) \end{aligned}$$

Figure 8. Future Analysis.

Assumptions: $ReadWriteConflict$: `label` $\times$ `label` $\rightarrow \{0, 1\}$ has value 1 if two program locations have conflicting accessing types, namely, if one of them is a write, and 0 otherwise.
Output: $Mark$: `label` $\rightarrow \{0, 1\}$ has value 1 if program point c can conflict with a program point in future M that is reachable from l, and 0 otherwise.
Initialization:
$\forall l \in M,\ Mark_{in}(l) := 0$

Analysis:

$$Mark_{out}(l) := \bigcup_{(l,s)\in F_M} (Mark_{in}(s))$$

$$Mark_{in}(l) := \begin{cases} 1 & \exists x \in (Access(l) \bigcap PT(c)) \text{ and } ReadWriteConflict(l, c); \\ Mark_{out}(l) & otherwise. \end{cases}$$

$$Access(l) := \begin{cases} \bigcup_{T\in PT(m)} SumAccess(T) & \texttt{l: future (m(...))}\ or\ \texttt{l: m(...);} \\ PT(l) & otherwise. \end{cases}$$

$$SumAccess(M) := \bigcup_{l\in M} Access(l)$$

Figure 9. Given a program point c and a future M of c, mark the statements in M that may reach an access that may conflict with c.

as a future is spawned at ***L1***. $Spawn_{in}(\textbf{\textit{L5}}) = \{\textbf{\textit{L1,L2}}\}$ since two futures are spawned at ***L1*** and ***L2***. $Spawn_{in}(\textbf{\textit{L6}}) = \{\textbf{\textit{L1}}\}$ because ***f2*** at ***L5*** must point to ***L2*** and thus ***L2*** is claimed. Note that ***L5*** does not add to its *Claim* set as it does not claim an external future. During the top-down phase, *Future* sets are computed in the main body and then propagated to `foo`. $Future_{out}(ENTRY_{foo}) = Future_{in}(\textbf{\textit{L1}}) \bigcup Future_{in}(\textbf{\textit{L2}}) = \{\textbf{\textit{L1}}\}$. As the local *Spawn* sets are all empty, according to the dataflow equation for $Future_{out}$, each point in `foo` has the same final result $Future_{out} = \{\textbf{\textit{L1}}\}$.

3.2 Concurrency Control Primitives Insertion

The next step of our analysis computes conflicting memory accesses and inserts ***allowed*** and ***grant*** barriers accordingly. Recall that the critical invariant that must be satisfied is that *a statement that accesses a shared object has to wait for the completion of all conflicting accesses in concurrently executing futures.* The set of parallel futures at any given point is computed using the analysis described above.

More precisely, our analysis inserts an ***allowed*** barrier for each shared access *c*, whose *Future* set is not empty. Given a method *M* that is pointed to by an element in $Future(c)$, the analysis described in Figure 9 is used to figure out program points to insert ***grant*** barriers in *M*.

The analysis marks statements in *M* that may perform an access which conflicts with *c* along some path. The analysis is a backward dataflow analysis – a statement is marked if any of its successors are marked. Hence, the confluence operator is defined as set-union. A conflicting access is defined as a non-empty intersection between the points-to set of *c* and the points-to set of the current statement in which both statements are not reads. Note that if the statement is a method call, the set of accesses of the statement is the union of the summaries of all the methods that can be applied at that point.

Given the analysis, the instrumentation rule is that *if a statement is not marked but its control flow predecessor is marked, this statement is the earliest point along a path to promise no conflict until the end of the future, and thus a* ***grant*** *barrier can be inserted immediately before it*.

Example. Consider the example in Figure 7. Two ***allowed*** barriers are inserted before the shared accesses at ***L7*** and ***L8***. Let us focus on inserting ***grant*** barriers accordingly for ***L7***. Since $Future_{out}(\textbf{\textit{L7}}) = \{\textbf{\textit{L1}}\}$ and method `foo` is the only future spawned at ***L1***, ***L7*** may run in parallel with other operations in `foo`. Hence, the analysis in Figure 9 is applied to ***L7*** and `foo`. Statements ***S1***, ***S2***, ***S3***, and ***L7*** are not marked since they do not conflict with the read of shared variable `x` at ***L7***. Statement ***L8*** is marked because the read of `x` at ***L7*** conflicts with the write of `x` at ***L8***. Since ***L7*** is not marked and its predecessor, the switch statement, is marked (because its successor at ***L8*** is), according to the insertion rule, a ***grant*** barrier is thus inserted before ***L7***. Similarly, another ***allowed*** is inserted after the write and before ***S2***. The resulting instrumentation related to ***L7*** is exactly the same as that presented in Fig. 4.

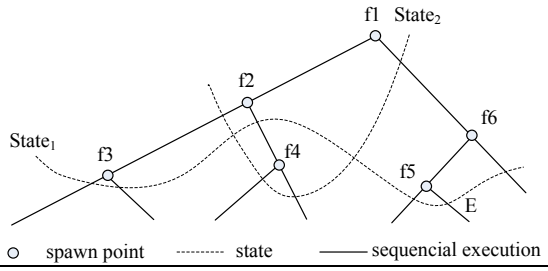

Figure 10. Dynamic Spawn Graph.

3.3 Runtime

An efficient lightweight runtime is critical for our technique to work well in practice. As the safety property demands a shared access not be discharged until all concurrently executing futures have completed their conflicting accesses, an ***allowed*** barrier on a condition ensures that all futures in the logical past have granted the condition. A naive implementation would maintain a sequentially ordered list of all futures, and would consult this list prior to any shared access to determine if there is the possibility of conflicts from any future in the logical past of the one performing the access.

While simple, such a design is highly inefficient. Consider the graph shown in Fig. 10 that reveals the spawning relationship between futures. We omit points where futures are claimed from the figure. A node denotes a future spawn site, and the left and right edges represent the spawned future and its continuation, respectively. An edge represents a segment of sequential execution. For example, edge `E` belongs to future `f6`; more precisely, it occurs as the continuation of the future spawned by `f6` denoted as `f5`. Thus, the actions performed by `f1, f2, f3, f4` (and their associated continuations) as well as the operations performed by `f5` constitute `E`'s logical past. Note that `f6`'s continuation is not included in the set; it logically executes *after* `E` completes. However, since the interleaving of futures and the order in which they complete is dependent on scheduling decisions that are not manifest in the program, at any particular point of time, a future in the logical past of a statement may not yet have even been spawned.

In Fig. 10, dotted edges are used to represent possible states of program execution. For instance, $\texttt{State}_1$ represents a possible runtime image, in which `f1, f2, f3, f5` and `f6` are spawned but not `f4`. If at this moment, `E` is about to execute a shared access, a runtime check on the futures that have currently been spawned would omit `f4` and lead to a safety violation.

In addition to these correctness concerns, maintaining the list of active futures corresponding to a logical sequential order may be expensive because the order in which futures are spawned and scheduled may be completely different from the desired sequential order. Consider $\texttt{State}_2$, in which `f1`, `f2`, and `f4` are active. Either `f3` or `f6` could be the next spawned future, even though `f3` precedes `f6` in the desired sequential execution order.

The main problem with our naive design is the maintenance of a central image of active futures. Thus, rather than trying to enforce a global time ordering, our runtime distributes the ordering among individual threads responsible for executing futures. Each thread maintains a local image of active futures in the order in which they were created within this thread. This image is propagated from a parent to its future children. For example in Fig 10, the main thread spawns `f1`. During its execution, denoted by the edge `f1` $\rightarrow$ `f6` in the main thread, `f1` is the only active future in the thread's local image. In other words, although `f1` further spawns `f2, f3` and `f4`, the main thread is unaware of these actions. From its point of view, these futures are indistinguishable from ordinary sequential method calls and the execution of the ***grant*** barriers performed by their ancestor `f1` promise the safety property for accesses it performs. More precisely, `f1` does not grant the condition for a shared access until all its future children containing these conflicting accesses indicate it is safe to do so. Our static analysis helps ensure this behavior. When the main thread spawns `f6`, the thread responsible for executing this future inherits the image of active futures from its parent (the main thread) and thus has the active list {`f1`}. When `f6` spawns `f5`, the active list is changed to be {`f1`, `f5`}. Note that the order these active futures are spawned follows the sequential order.

The pseudocode of our runtime is presented in Fig. 11. Methods ***spawn*** and ***get*** maintain the list of local active futures. When a future is spawned, the current thread acquires a global id for the

Table 1. A Sample Run with The Runtime Support.

Timestamp	*main* AL=ϕ	*f1* AL=ϕ	*f2* AL=$\{f1\}$	*f3* AL=$\{f1, f2\}$	*f4* AL=$\{f1, f2, f3\}$	condition a
1	f1= new future (foo...)					0...00
2	f2= new future (foo...)	S0				0...00
3	f3= new future (foo...)	***grant(a)***	S0			0...01
4	f4= new future (foo...)	***allowed(a)***	***grant(a)***	S0		0..011
5	S4	R_x	***allowed(a)***		S0	0..011
6	...	S1	R_x		***grant(a)***	0..01011
7	...	S3	S1	W_x	***allowed(a)***	0..01011
8	...		S3	***grant(a)***		0..01111
9	...			S2	R_x	0..01111
...	...			...	...	...

```
    void spawn (Future f )
1.  {
2.     synchronized (gidLock) {
3.        f.id= gid ++;
4.     }
5.     active.add(f.id);
6.     need2Wait.setAll( );
7.  }
8.
9.
10.
11. void grant (Condition c)
12. {
13.    c.setbit (getCurrentFuture( ).id);
14. }
```

```
    void get (Future f)
15. {
16.    active.remove(f.id);
17. }
18.
19. void allowed (Condition c)
20. {
21.  if (!need2Wait[c]) return;
22.  l=active.length( );
23.  for  (i=0; i<l; i++)
24.    if (!c.getbit (active.get(i).id) ) {
25.          getCurrentThread( ).yield( );
26.          i--;
27.          continue;
28.    }
29.    need2Wait[c]=FALSE;
     }
```

Figure 11. Runtime.

```
    foo (… , int op, int n)
1.  {
2.    if (!n) return;
3.    switch (op) {
4.      case 1:
5.        allowed(a);
6.        Rx;
7.        S1;
8.      case 2:
9.        Wx;
10.       S2;
11. }
12. foo (x, op, n-1);
13. grant(a);
    }
```

Two futures with the form of foo (…,2) are spawned

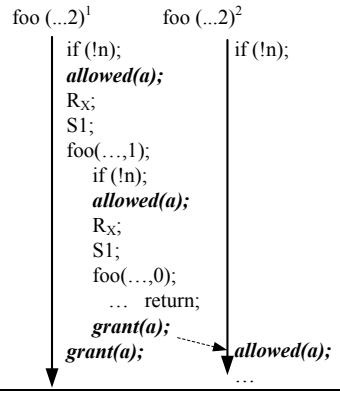

Figure 12. The Problem of Infeasible Paths.

new future and adds it to the list. The flag `need2Wait` serves as a shortcut to avoid redundant permission checking (see below). Each shared access has an associated condition which contains a bitvector to maintain the status of all spawned futures. If a future has performed the conflicting accesses, it calls the ***grant*** barrier on the associated condition to set its bit. In the implementation of the ***allowed*** barrier, `need2Wait` is first checked; if it is false, the thread has earlier blocked on this condition and further blocking is not necessary. If true, the operation scans the condition's bitvector and sees if the bits corresponding to the active futures have been set. To avoid redundant checking, it stalls at individual unset bits.

Example. Consider the execution in Table 1 with the instrumentation shown in Fig. 4. The main thread spawns four futures at timestamps 1, 2, 3 and 4. The futures have the active list (AL) as ϕ, `{f1}`, `{f1, f2}`, `{f1, f2, f3}`, respectively. Future `f1` sets the first bit of condition `a` at timestamp 3. Before `f1` discharges the read, it executes the ***allowed*** barriers, which returns right away as `f1`'s active list is empty. Future `f2` sets the second bit of `a` at timestamp 4. Because the first bit is set, `f2`'s execution of ***allowed*** at timestamp 5 can return immediately without blocking. In contrast, the ***allowed*** barrier performed by `f4` is blocked until timestamp 9 because `f3` is a member of `f4`'s active list and `f3` does not notify until it has completed its write. The safety property is thus guaranteed.

4. Limitations

Because a substantial component of our approach is based on a static analysis, our safety guarantees are conservative.

For example, our technique can fail to sometimes exploit latent concurrency due to the presence of infeasible paths; these paths can confound the analysis, and cause notification actions to be inserted later than necessary. Consider the example in the left hand side of Fig. 12. It is very similar to our earlier example in Fig 1 except that `S3` is instantiated with a recursive call to method `foo` itself and `S0` is replaced with a recursion termination check. This code fragment summarizes the control structure in a frequently invoked method in OO7 [6], one of the benchmarks we consider in the following section. The instrumentation that allows discharging the read access is highlighted in the figure. As we can see, our analysis puts the ***grant*** after the recursive call, even though it would be safe to insert it at the same points as in Fig 4, namely, before R_x and after W_x. The reason is that the analysis is not capable of determining that variable `op` does not change during recursive calls so that once the method reaches the read access, it is infeasible for the execution to reach the write access; subsequent recursive calls are guaranteed to not change the `op` predicate. As a result, the analysis conservatively assumes that both the read and write accesses may occur inside the recursive call regardless of the path taken, which prevents the ***grant*** from being hoisted. Suppose two futures with the recursion depth `n=2` are spawned; their interleavings are depicted on the right hand side of Fig. 12. As we can see, the read access of the second future has to wait until the end of the series of recursive calls performed by the first, effectively sequentializing execution. As part of our future work, we plan to integrate a theorem proving engine into our static analysis to alleviate this issue. Currently, our implementation requires programmers to explicitly indicate infeasible paths so they are not considered by the analysis.

5. Experiments

For our experiments we annotated a set of Java benchmarks with futures and then used our technique to statically insert concurrency

Table 2. Benchmark Characteristics.

Benchmark	Lines of Code	Static Barrier #	Dynamic Barrier #	Futures Spawned
series	750	0	0	2
mc	3460	0	0	60000
lusearch	34371	7	176	32
OO7	2080	21	3350	500
TreeMap	1850	11	2080	400

control to guarantee safety. In general, annotating the benchmark programs required only minimal understanding of the code. Our experiments compare the performance of the transformed benchmark to its sequential version.

Our benchmarks were compiled with Sun's `javac` 1.5.0 compiler. We used the future implementation in the `java.util.concurrent` package. We used Soot version 2.2.4 for the analysis and transformation. In particular, the SPARK [20] flow-insensitive, context-insensitive, and field-sensitive may points-to analysis was used. We also used Soot to identify objects created in a method that never escape as local objects. The analysis does not insert concurrency barriers for field accesses of such objects. The experiments were run on Java HotSpot(TM) 64-bit Server VM 1.5.0. Our hardware platform is an 1.8GHz Dual Core AMD Opteron(tm) Processor 865, with 8 dual-core CPU's, and 32GB of RAM running Linux kernel version 2.6.9-34. We report results on 2 dual-core processors (4 cores) and 8 dual-core processors (16 cores).

5.1 Benchmarks

As shown in Table 2, our experiments were performed on two Java Grande benchmarks [27], namely `series` and `mc`, `lusearch` from the DaCapo benchmark suite (version 2006-10) [3], OO7 [6], and a Java tree map implementation that uses a red-black tree[1]. The Java Grande benchmarks are representative of computationally intensive programs where one can easily identify code sections or methods where concurrency may be achieved. `Lusearch` is a multi-threaded benchmark with explicit concurrency control which we transform to use futures. OO7 represents a complex irregular database application, and the tree-map benchmark performs operations on a dynamic mutable data structure. Because both OO7 and tree-map manipulate complex data structures, exhibit aliasing, and complicated sharing across procedure boundaries, they are less amenable to traditional loop-based parallelization techniques.

Three out of the five benchmarks considered (`series, mc` and `TreeMap`) were annotated by a simple syntactic inspection of the code, with no deeper understanding of the semantics of the benchmark. `Lusearch` was annotated by removing threads and explicit synchronization primitives and replacing them with futures. Annotating OO7 required a modest understanding of the underlying data and control structure (see Section 4). All of the rewrites of sequential programs required either changing existing method calls into future calls or moving the body of a loop into a future call with no further changes to the control flow or semantics of the application. In particular, our parallelizations did not consist of partitioning arrays into sub-arrays or any other restructuring of the code. Not all benchmarks we considered were amenable to future insertion. We originally considered two other Java Grande benchmarks, `moldyn` and `raytracer`, but found that they are not amenable to futures without a significant rewrite of the code. For example, the `moldyn` benchmark was naively translated from C code, and thus all variables, including loop counters, were declared as class fields instead of local variables. Therefore, any simple translation from a method to a future requires that these fields be considered as shared variables, and thus our technique would introduce a large number of ***allowed/grant*** barriers, making the execution essentially sequential.

Table 2 also shows the number of lines in the original benchmark, the number of static and dynamic ***allowed/grant*** barriers inserted by the analysis, and the number of futures spawned in each execution. More details are provided below.

5.1.1 Java Grande

The `series` benchmark performs Fourier coefficients computation. The code calculates two terms for each coefficient by calling the method `TrapezoidIntegrate` twice. We annotated the two calls to `TrapezoidIntegrate` to be spawned as futures. Then we claimed the result of the two futures. Our analysis determines that there is no need to insert ***allowed/grant*** barriers as the method does not access any shared data.

The `mc` benchmark is an implementation of a Monte Carlo Simulation. Our future version was a straightforward rewrite such that the body of the main loop was spawned as a future. The future initializes a local price stock object and performs some computation on the object. After futures are spawned in the main loop, they are claimed in another loop and their results are added to the global vector as in the sequential benchmark. Our analysis recognizes that each future performs calculations on local objects and therefore does not insert any ***allowed/grant*** barriers.

5.1.2 Lusearch from DaCapo

The `lusearch` benchmark is a text search tool that is part of the DaCapo benchmark suite. The benchmark creates an index and searches for keywords in large text files. The original benchmark creates 32 threads and each thread searches for approximately 3500 distinct words.

The threads create local search indexes and operate on local objects until they complete their search. Once a thread has completed searching the thread updates an integer field in the shared parent object that spawned the threads. Access to this shared variable is protected by a monitor on the parent object.

In our experiments we replaced threads with futures and removed all synchronization. To measure the computational parallelism exploited by futures, we isolated the computation by loading the query files and the search indexes into memory before performing the searches. Our analyses injected seven synchronization calls. The calls were injected for field accesses of the parent class from the futures. These accesses were the same as those protected by explicit concurrency control mechanisms in the original program.

5.1.3 OO7

The underlying data structure in OO7 consists of a set of graphs and indices that are intended to simulate an object-oriented database system. Each graph has a multi-level complex index to leaves which contain several *composite parts*, each composed of a *document* and links to a graph of *atomic parts*. The execution of OO7 is dominated by data traversals. A traversal chooses a single path through the index graph and at the leaf level chooses a fixed number of composite parts to visit. This number is configurable; we used 16 composite parts in our experiments.

For our experiments we used a single graph with seven levels and each intermediate level has three children. There are a total of 500 composite parts at the leaf level, each corresponding to a graph of 100000 atomic parts. We rewrote the sequential traversals so that the composite parts were traversed concurrently. Since the data structure contains 500 composite parts, the execution of the future version spawns 500 future.

5.1.4 MultiTreeMap

The final benchmark performs `put` and `get` operations on a red-black tree implementation of a map. The MultiTreeMap [1] is a TreeMap much like `java.util.TreeMap` except it allows duplicate keys to be entered. Our benchmark allows the user to configure the percent `get` operations and the percent `put` operations.

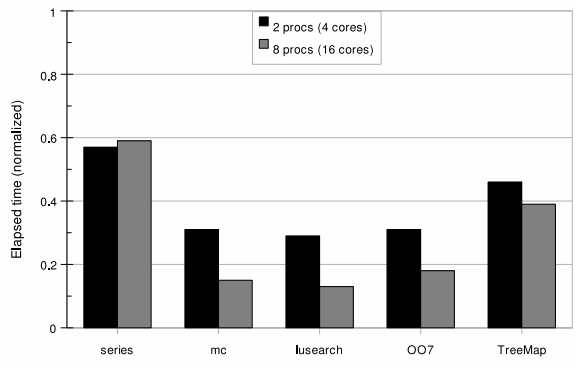

Figure 13. Elapsed time (normalized)

The sequential version of the benchmark loops 400 times and for each iteration of the loop randomly selects operations based on the configured parameters. For the future version, we simply spawned each iteration of the loop as a future.

5.2 Results

Figure 13 reports the elapsed time for the future version of each benchmark normalized against the average elapsed time of the sequential version of the benchmark for 2 processors (4 cores) and 8 processors (16 cores).

We also ran experiments to measure the fundamental overhead of our runtime. The overhead was calculated by running the future version of each benchmark such that the calls to spawn future did not spawn any threads, and the futures were run synchronously as ordinary methods, but with the synchronization overhead caused by ***allowed/grant*** primitives. The overhead of our approach ranged from less than a percent to roughly 3%.

5.2.1 Java Grande

The reported times for the Java Grande benchmarks are the arithmetic mean of 4 runs of each benchmark.

The speedups for the future version of the `series` benchmark were approximately 1.75 and 1.70 over the sequential version for 2 processors (4 cores) and 8 processors (16 cores) respectively. This result is to be expected since only two futures are spawned in the future version, thus at most 3 cores are being used (2 for the futures and 1 for the continuation). The computation that is not parallelized makes the speedup less than 2.

The future version of the `mc` benchmark demonstrates speedups of 3.23 and 6.76 over the sequential version for 4 cores and 16 cores respectively. The futures in the `mc` benchmark perform local computations concurrently and then the continuation claims the results. The continuation sequentially updates a global vector with the results of the local computations.

5.2.2 Lusearch from DaCapo

The reported results were obtained by running `lusearch` on DaCapo's default input for this benchmark 4 times and taking the arithmetic mean. We report speedups of 3.42 and 7.45 for 4 cores and 16 cores respectively. The future version of the benchmark allows futures to execute concurrently until the first access to the shared integer field in the parent class. Our analysis and runtime ensures that the futures perform race-free accesses to the shared field in the order the futures were spawned.

5.2.3 OO7

The reported results were for 70% shared read traversals and 30% shared write traversals. Our results show approximately 3.85 and 6.67 speedup for 4 cores and 16 cores respectively for this benchmark. As our analysis decides all traversals access the same abstract data, a write traversal in the future version of the benchmark requires that all previous traversals, reads and writes, complete before continuing. Furthermore, it blocks traversals that follow it in the logical time order. Thus, the improvement of runtime results from concurrent read traversals of composite parts.

5.2.4 MultiTreeMap

The analysis forces the `get` operation to wait for any previous `put` operations to complete, but does not need to wait for any previous `get` operations since `get` operations are read-only. The `put` operation, on the other hand, will need to wait for all previous `get` and `put` operations to complete. Thus `get` operations can run concurrently, but any `put` operation will force all previous operations to complete before running. Therefore, the results mainly depend on the distribution of `get`/`put` operations, which is roughly 4:1 in our experiment. As shown in Fig 13, the resultant speedups for 4 and 16 cores are 2.17 and 2.56, respectively.

6. Related work

Automatic Program Parallelization. Traditional automatic program parallelization exploits concurrency across loop iterations using array dependence analyses [2, 10]. In programs which exhibit more complex dataflow and control-flow mechanisms, these techniques are not likely to be as effective. Parallelizing general sequential programs in the presence of side-effects has been explored in the context of Jade [25]. A Jade programmer is responsible for delimiting code fragments (tasks) that could be executed concurrently and explicitly specifying invariants describing how different tasks access shared data. The run-time system is then responsible for exploiting available concurrency and verifying data access invariants in order to preserve the semantics of the serial program. Commutativity analysis [24] exploits coarse-grained parallelism for object-oriented programs using symbolic execution to decide if two operations can commute and thus run in parallel.

Recent work by Harris and Singh [13] describe a profiling infrastructure for Haskell that can be used to identify transparently opportunities for parallelism in Haskell programs. Their approach profiles compiled thunks to identify promising sources of parallelism, rewrites the program to speculatively execute thunks identified as good candidates for concurrent execution, and modifies the language runtime to monitor when speculative threads might perform unsafe I/O operations; these threads are suspended, and resumed only when necessary.

Recently, speculative parallel execution through thread level speculation [21, 16, 28, 31] and transactions [15, 12, 14] have been proposed. These techniques speculatively execute concurrent threads and revoke execution in the presence of conflicts. Kulkarni et al [19] present their experience in parallelizing two large-scale irregular applications using speculative parallelization. Compared to these techniques, our method does not require programmers to significantly rewrite the program to spawn threads or define concurrency control, does not hinge on complex hardware or runtime support, and thus enables portability and applicability.

Futures. Futures were first introduced in Multilisp [11] as a high level concurrency abstraction for functional languages. The semantics of futures [9] and their implementation [18, 23] have been well-studied in the context of functional languages like Scheme [17]. More recently, the Java 2 Platform Standard Edition 5.0. also includes support for futures; unfortunately, the specification makes no guarantees about safety. Safe futures [30] encapsulate futures within software transactions so that safety violations can be prevented or remedied. Implicit Parallelism with Ordered Transactions (IPOT) [29] proposes a concurrency abstraction similar to futures except that a spawned speculative subtask is only

allowed to spawn a unique successor. This restriction simplifies concurrency management and dependence tracking, at the cost of potential loss of parallelism. Most recently, an efficient implementation [8] of a similar parallel execution model is proposed for unsafe languages like C.

Static Interference Analysis. Our technique is also related to static analysis for interference. Rugina and Rinard [26] propose an interprocedural, flow- and context-sensitive pointer analysis for structured parallel programs written in *Cilk* [4]. Escape analysis [7] analyzes if an object escapes thread boundary. Boyland proposes a type system [5] that checks interference using the concept of linear capabilities. None of these approaches are designed with the specific semantics of futures in mind.

Related to our work is Autolocker [22], a tool that converts pessimistic atomic sections into lock-based code. Autolocker uses a whole-program type-based analysis to generate a dependency graph that is then used to insert lock operations that preserve necessary ordering. Lock insertion is similar to our injection of barriers to notify and block threads, but the structure of the two analyses are quite different given the differences in semantics between futures and atomic sections.

7. Conclusions

Quasi-static scheduling is a new technique designed to facilitate the migration of sequential program to multi-core platforms. The essence of the technique lies in an interprocedural summary-based program analysis that computes dependencies among concurrently executing threads. Potential violations are prevented using lightweight barriers. We have explored the use of quasi-static scheduling in the context of Java futures, a concurrency abstraction for Java that allows the specification of asynchronous method calls. Experimental results reveal that our approach can allow future-annotated programs to avail of additional computing cores, without sacrificing safety, or requiring complex concurrency control.

Acknowledgements

This work is supported in part by the National Science Foundation under grants CNS-0720516, CNS-0708464, CCF-0701832, CNS-0509387, and by a grant from the Intel Corporation.

References

[1] Multitreemap. http://sourceforge.net/projects/multitreemap/.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *Journal of Parallel Distributed Computing*, 37(1):55–69, 1996.

[5] John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

[6] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *SIGMOD*, pages 12–21, 1993.

[7] J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[8] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.

[9] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *POPL*.

[10] Mary W. Hall, Brian R. Murphy, Saman P. Amarasinghe, Shih-Wei Liao, and Monica S. Lam. Interprocedural analysis for parallelization. In *LCPC*, pages 61–80, 1995.

[11] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[12] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[13] Tim Harris and Satnam Singh. Feedback Directed Implicit Parallelism. In *IFCP*, pages 251–264, 2007.

[14] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC*, pages 92–101, 2003.

[15] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[16] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI*, pages 59–70, 2004.

[17] R. Kelsey, W. Clinger, and J. Rees. Revised 5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998.

[18] David Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A High-Performance Parallel Lisp. In *PLDI*, pages 81–90, 1989.

[19] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[20] O. Lhotak. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, 2002.

[21] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In *PPOPP*, pages 158–167, 2006.

[22] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *POPL*, pages 346–358, 2006.

[23] Rick Mohr, David Kranz, and Robert Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP*, pages 185–197, 1990.

[24] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *IPPS*, pages 14–22, 1996.

[25] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, 1993.

[26] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, 2003.

[27] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel java grande benchmark suite. In *ACM Supercomputing*, page 8, 2001.

[28] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–12, 2000.

[29] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *PPOPP*, pages 79–89, 2007.

[30] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *OOPSLA*, pages 439–453, 2005.

[31] L. Rauchwerger Y. Zhan and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *HPCA*, pages 162–173, 1998.