Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java

Lingli Zhang Chandra Krintz Priya Nagpurkar
Computer Science Department
University of California, Santa Barbara
{lingli_z,ckrintz,priya}@cs.ucsb.edu

Abstract

In this work, we investigate the implementation of futures in Java J2SE v5.0. Java 5.0 provides an interface-based implementation of futures that enables users to encapsulate potentially asynchronous computation and to define their own execution engines for futures. Although this methodology decouples thread scheduling from application logic, for applications with fine-grained parallelism, this model imposes an undue burden on the average users and introduces significant performance overhead.

To address these issues, we investigate the use of lazy futures and offer an alternative implementation to the Java 5.0 approach. In particular, we present a directive-based programming model for using futures in Java that uses annotations in Java 5.0 (as opposed to interfaces) and a lazy future implementation to significantly simplify programmer effort. Our directive-based future system employs novel compilation and runtime techniques that transparently and adaptively split and spawn futures for parallel execution. All such decisions are automatic and guided by dynamically determined future granularity and underlying resource availability. We empirically evaluate our future implementation using different Java Virtual Machine configurations and common Java benchmarks that implement fine-grained parallelism. We compare directive-based lazy futures with lazy and Java 5.0 futures and show that our approach is significantly more scalable.

1. Introduction

As multi-processor computer systems become increasingly ubiquitous, it is vital that popular, high-level, programming languages enable easy-to-use and efficient parallel programming. One simple and elegant construct that programmers can use to identify potentially asynchronous computation, is the *future*. Futures were introduced in Multilisp [11, 22], and are supported by many modern programming languages using a variety of implementations (e.g. Java J2SE 5.0 [14], X10 [7] and Fortress [1]).

In this work, we investigate an implementation of futures for Java that is both easy to use and that enables high performance and scalability, especially for applications with significantly fine-grained parallelism. The current Java future APIs [14] employ a *Executor-Callable-Future* programming model. With this model, users encapsulate a computation that can be potentially evaluated in parallel using a `Callable` object and submit the object to an `Executor` for execution. The Executor creates and returns a `Future` object to the initiating thread with which it can later access the return value computed by the computation. The initiating thread immediately executes the code that follows this process, i.e., the continuation. The Java API provides several general Executors that implement different thread scheduling policies for futures and users can implement their own, customized versions via this interface. This decoupling of thread scheduling from application logic greatly simplifies parallel programming in Java.

The disadvantages of this approach are two-fold: the programmer must create the encapsulating objects for every potential future, and is burdened with future scheduling decisions. Since Executors execute in the application context, they have no control over, or feedback from, the services internal to the Java Virtual Machine (JVM), e.g., compilation, thread scheduling, performance monitoring, etc. Feedback from these services, such as program behavior, future granularity, and underlying resource availability, are key for deciding when to spawn a future (by spawning a thread) and when to execute it inline (directly). However, employing this information effectively is complex and different for every system.

To simplify the programming with futures in Java and to enable automatic and adaptive spawning of futures, we present Directive-based Lazy Futures (DBLFutures) in this paper. DBLFutures are inspired by parallel programming models for other languages that employ keywords or directives to identify parallel computations [5, 18, 7, 1, 20]. Using the DBLFuture programming model in Java, users annotate the variable declarations of all variables that store the return value from a function that can be potentially executed concurrently with `@future` directives. Using DBLFutures, the parallel version of a program is the same as the serial version with annotations on a subset of variable declarations.

Our DBLFuture implementation moves future scheduling into the JVM. To enable this, we extend Lazy Java Futures (LazyFutures) [29], a JVM future implementation based on Lazy Task Creation (LTC) [19]. LazyFutures first executes potentially concurrent computations inline and then spawns them via runtime stack splitting during thread switching and scheduling if the system estimates that the remaining execution time of the future will amortize the cost of splitting. To estimate future execution time, LazyFutures build upon the performance monitoring service that is common to most JVM implementations.

Our extensions to this system include a set of compiler and runtime modifications that support our directive-based future specification, yet spawn futures lazily with low overhead. Key to the efficacy of our extensions, is our avoidance of all object creation for sequentially invoked, potential futures. When the sampling system identifies a future as long running and there are sufficient processor resources, the system splits the thread stack into two, creates the future object, and executes the future and continuation in parallel. Our compilation system produces code (inline, not duplicated) that checks whether the future was spawned or directly executed and handles the return value storage and first access appropriately and transparently.

We empirically evaluate our system using a number of JVM configurations an Java programs that implement different amounts of fine-grained parallelism. Our results show that our system enables speedups of 2$\sim$11 times over Lazy and Java 5.0 future implementations, is significantly more scalable, and imposes very low overhead.

2. Java Futures

The Java 5.0 release provides several new interfaces and library support of concurrent programming. One significant extension is the *Executor-Callable-Future* programming model. In this model, programmers encapsulate a computation that can be safely evaluated in parallel in a `Callable` object and submit the object to an `Executor` object for execution scheduling. The Executor returns a `Future` object that the current thread can use to query the results that the computation eventually returns; the current thread initiates execution of the continuation. The Java 5.0 API provides several implementations of `Executor` with various scheduling strategies. Moreover, programmers can implement their own Executors with which they can customize scheduling decisions. We refer to this programming model as *J5Future* in this paper. Figure 1(a) shows a simplified program for computing the Fibonacci sequence (*Fib*) that uses the Executor-Callable-Future model and its Java 5.0 interfaces.

The J5Future programming model is simpler than a thread-based model since it decouples thread scheduling from application logic. However, since this model requires that users wrap all future computation into an object, significant programmer effort may be required to convert serial versions of programs to parallel versions. In addition, given the multiple levels of encapsulation, this model consumes significant memory (which then must be managed by garbage collection) for each future.

The second drawback of this model is that it places the burden of scheduling futures on the user. The default Executors are effective for simple cases but do not consider the granularity of computation or underlying resource availability – both of which are vital to achieving scalability and high-performance but are very difficult to extract accurately at the library level. As a result, especially for applications with fine-grained parallelism, naive Executors, e.g., those implemented using a thread-pool to execute each submitted future, can severely degrade performance and scalability.

Users can create their own Executors and/or hardcode thresholds that attempt to identify when to spawn (and amortize the cost of spawning) or inline futures. However, this requires expert knowledge about the dynamic behavior of the program and the characteristics (the spawn cost of futures, and the compilation systems, processor count and availability, etc.) of the platform on which the application ultimately executes. Moreover, regardless of the expertise with which the scheduling decisions are made, this model, since it is implemented outside and independent of the runtime, is unable to exploit the services (recompilation, scheduling, allocation, performance monitoring) and detailed knowledge of the system and program that the execution environment has access to.

```
public class Fib
  implements Callable<Integer>
{
  ExecutorService executor = ...;
  private int n;

  public Integer call() {
    if (n < 3) return n;
    Future<Integer> f =
      executor.submit(new Fib(n-1));
    int x = (new Fib(n-2)).call();
    return x + f.get();
  }
  ...
}
```

(a) Futures in Java 5.0

```
public class Fib
  implements Callable<Integer>
{
  private int n;

  public Integer call() {
    if (n < 3) return n;
    LazyFutureTask<Integer> f =
      new LazyFutureTask(new Fib(n-1));
    f.run();
    int x = (new Fib(n-2)).call();
    return x + f.get();
  }
  ...
}
```

(b) Object-oriented lazy futures

```
public class Fib
{
  public fib(int n) {
    if (n < 3) return n;
    @future int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
  }
  ...
}
```

(c) Directive-based lazy futures

Figure 1. Different programming models for futures in Java

To exploit the services and knowledge of the Java Virtual Machine (JVM) and to reduce the burden of using Executors in Java 5.0, we previously have introduced *Lazy Futures* for the Java language and virtual machine [29] (to which we refer to as LazyFuture throughout).

The LazyFuture implementation is inspired by a technique originally proposed by Mohr et al. [19], called *lazy task creation* (LTC). LTC initially implements all futures as function calls. The system then maintains special data structures that enable spawning of the continuation. When there is an idle processor available, the idle processor steals continuation from the first processor and executes it in parallel with the future. Similar techniques are employed for many different languages to support fine-grained parallelism [21, 9, 10, 25]. The LazyFuture implementation for Java employs volunteer splitting as opposed to work stealing and it couples dynamic information about computation granularity with underlying resource availability to make scheduling decisions.

LazyFutures use an abstraction called the `LazyFutureTask` which extends the Java 5.0 FutureTask. Users create a `LazyFutureTask` object for each potentially asynchronous computation and invoke its `run()` method directly (in a way similar the traditional Java thread model). Figure 1 (b) shows the LazyFuture implementation of *Fib*.

The LazyFuture-aware JVM recognizes this `run()` method (in each `LazyFutureTask`). The JVM initially inlines the computation on the current thread (i.e. it does not spawn the future). The system then monitors the computation to estimate its granularity and the underlying resource availability to determine when spawning will result in improved performance. LazyFutures leverage adaptive optimization and the low-level program and system information to which a JVM has access. The system estimates granularity using feedback from the sampling system common to JVMs that identifies "hot", i.e., long running, methods. The system then splits the runtime stack of a thread that is executing a hot future into two, the future and the continuation, and relies on the internal Java threading system to enable efficient scheduling. Since this process happens only when a thread switch occurs (approximately every 10ms in our JVM), the overhead of monitoring methods is hidden and thread synchronization is unnecessary and avoided.

3. Directive-based Lazy Futures

The LazyFuture model follows an interface-based approach that is similar to (yet more efficient than) J5Futures. As a result, it inherits similar programmer productivity and performance disadvantages. Using the interface-based approach, users must employ object encapsulation of futures, and thus, incur memory allocation and management overhead. In addition, the coding style using this methodology imposes an extra burden on the programmer and causes source code to be longer and less readable in order to specify and use the interface. To address these limitations, we propose a new implementation of futures in Java that we call *Directive-based Lazy Futures (DBLFutures)*.

Our DBLFuture implementation builds upon and extends LazyFutures to improve the ease-of-use of future-based parallelism in Java as well as performance and scalability. DBLFutures exploit the Java language extension for annotations (JSR-175 [15]). Annotations are source code directives that convey program metadata to tools, libraries, and JVMs; they do not directly affect program semantics. In particular, we introduce a future annotation (denoted `@future` in the source code) for local variables. Users employ our future directive to annotate local variables that can be used as placeholders of results returned by function calls that can be potentially executed concurrently by the system. If a function call stores its return value to a annotated local variable, it is identified as a future function call. Note that in our system, the scope of future annotations is within the method boundary. If the value of a local variable with the future annotation is returned by the method, the future annotation will not be returned with the return value. Figure 1(c) shows the implemented *Fib* program using this model.

Our DBLFuture model avoids creation (and thus, user specification) of `Callable`, `Future`, `LazyFutureTask`, or other objects when the future is inlined (executed sequentially) by the system. As such, we avoid the memory allocation, memory management, and extra source code required by previous approaches. With this model, users easily specify computations that can be safely executed in parallel with minimal rewriting of the serial programs. This programming methodology also provides the JVM with the flexibility to implement potentially concurrent code regions as efficiently as possible. Note that with our current implementation of DBLFutures, the JVM makes efficient spawning decisions automatically, but the users are still responsible to ensure the safety of concurrent execution.

Our DBLFuture-aware JVM recognizes the future directive in the source and implements the associated calls using a set of LazyFuture extensions and compiler techniques. First, the future directive in the source is saved as a method attribute in the bytecode. The class loader of our DBLFuture-aware JVM recognizes this attribute and builds a future local variable table for each method, which contains the name, index, and bytecode index range of each future local variable. Our Just-In-Time, dynamic compiler consults this table during compilation.

Initially, the JVM treats every future call as a function call, and executes the code on the runtime stack of the current thread. For each such call, the system also maintains a small stack that shadows the runtime stack for each thread, called the *future stack*. This future stack maintains entries for potential future calls only. Each entry contains metadata for the corresponding runtime stack frame of the future call that includes the location of the frame on the runtime stack and a sample count that estimates how long the future call has executed. The system uses this information to make splitting and spawning decisions.

Each DBLFuture shadow stack frame also contains the local variable index and the stack slot in the runtime stack of the caller of the future call that the compiler has allocated for this local variable. Our system employs this information to set up the future and continuation thread correctly upon a split and spawn.

3.1. Future Compilation

For LazyFutures, the `LazyFutureTask.run()` method is the only marker of potential future calls in the program. In addition, the process of storing the return value of a future call and accessing the value later on is explicitly coded in the application via implementation of the `run()` and `get()` methods of the `LazyFutureTask` class. The `LazyFutureTask` object serves as the placeholder of the computation result, and is always created regardless of whether the computation is inlined or spawned.

The LazyFuture compiler implements a small, inlined, and efficient, stub in the prologue and epilogue of the `run()` method. This stub pushes an entry onto the future stack at beginning of a future call, and pops the entry off of the future stack when exiting the future call. In addition, the return type of the `run()` method is void, so the address of the first instruction of the continuation is the return address of the run method. Thus, upon future splitting, the system can extract the return address from the runtime stack frame for the `run()` method, and use it as the starting program counter (PC) of the new thread (that will execute the continuation). The system sets the original return address to a stub that terminates the current thread when the future call completes.

DBLFutures require a somewhat more complex compilation approach. We maintain the future stack for every marked future call as is done for LazyFutures. However, we want to allow any method call to be specified as a potential future call if it can be executed safely in parallel. We also want to allow the same method definition to be used in both a future and a non-future context. The extant compilation strategy requires that we produce two versions of compiled code for every method that may be used in the future context, and insert stubs into the prolog and epilog of all such methods. This is not desirable since it causes unnecessary code bloat and compilation overhead. Instead, we expand the future call cites and insert future stack maintenance stubs before and after the call site of the future.

The store of the return value after the future call completes requires special handling. If the call is not split, the return value must be stored into the specified local variable. If the future is split and spawned, the return value must be stored into a placeholder (i.e. a Future object) for access by the continuation thread. To enable this, we add one word to every runtime stack frame, for a *split flag*. This flag is a bitmap of spawned futures indexed by the future local variable index in the bytecode local variable array. For example, if the future call associated with a local variable at index 1 is spawned, the JVM sets the second lowest bit of the flag to 1. The JVM checks this bit at two points in the code: (i) at the store of the return value and (ii) at the first use of the return value. We currently support 32 futures (64 for 64-bit machines) per method given this use of a bitmap. However, we can extend this by using the last bit to indicate when there are more futures, and storing a reference to a full-fledged bit-vector if so.

Our compiler always allocates a slot on the runtime stack for every future-annotated local variable. This slot holds different variable types at different times: before splitting, its type is the declared type of the local variable; after splitting, it holds a reference to a `Future` object which is created and set by the splitting system in the JVM; after its first use, its type becomes the declared type again. To ensure correct garbage collection (GC), the compiler includes this slot in the GC maps and the garbage collector dynamically decides whether it holds a reference or not using the split flag.

We compile the return value storage point to a conditional branch. If the split flag is set, the code stores the return value directly in the local variable slot on the stack. Otherwise, the code extracts the reference to the `Future` object from the same stack slot, and stores the return value into the `Future` object.

We similarly expand instructions that use the return value. If the split flag is set, the codes uses the value in the local variable slot on stack directly; otherwise, the code executes the `get()` method on the `Future` object that it extracts from this same slot (which will block if the return value is not ready yet). In this latter case, when the system eventually returns a value from a method via the `get()` method, it also stores the value in the slot (an thus, the slot at this point holds the type of the original local variable). If there are multiple use points, our compiler only converts the first one (the one that dominates the others) since all uses thereafter are guaranteed to access the value with the original declared type. In addition, our compiler will insert a fake use of the future value before the method exit point if there is no usage of the future to prevent it escaping the method boundary. That is, a method will wait for all futures that it spawns to finish before it exits.

Finally, we must set the starting PC of the continuation thread correctly. Logically, if a future is split, the continuation thread should start at the point in the code *immediately after* the point at which the return value is stored. Note, though, that this is not the return address of the future call any longer (as is the case for LazyFutures). To provide this information to the JVM splitting mechanism, we insert a fake instruction after the return value store instruction which we pin throughout the compilation process. At the end of compilation we remove this instruction; but, we put its PC and the index of the associated local variable into a map which we store with the compiled code and query during future splitting.

Bench- marks	Inputs size	future# (10^6)	BaseVM (secs)	PAOptVM (secs)
FFT	2^{18}	0.26	43.63	8.27
Raytracer	balls.nff	0.27	167.00	20.41
AdapInt	0-250000	5.78	102.45	28.84
Quicksort	2^{24}	8.38	54.10	8.77
Knapsack	24	8.47	106.75	11.88
Fib	40	102.33	673.40	29.09

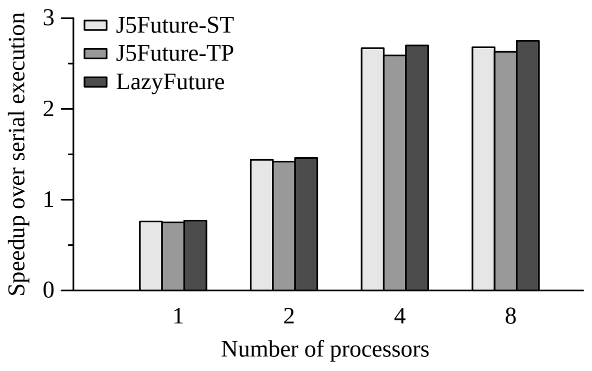
Table 1. Benchmark characteristics.

By extending a JVM, our DBLFutures implementation avoids complicated source or bytecode rewriting or multiple code versions and yet easily enables migration from inlined to concurrent execution. In addition, our system is able to mix future calls with normal calls naturally since we have access to the Java operand stack and local method state. Non-JVM implementations cannot do this easily. For example, Cilk and JCilk [5, 18] do not allow non-Cilk method to call a Cilk method at all since a non-Cilk method is not compiled with parallel support (fast and slow clones) and is not migratable.

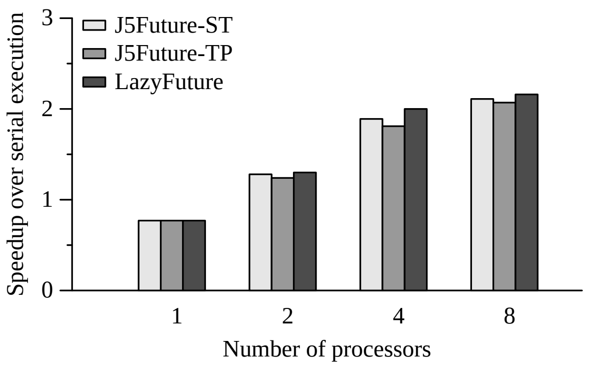
4. Experimental Methodology

We have implemented DBLFutures (as well as LazyFutures) in the popular, open-source Jikes Research Virtual Machine (JikesRVM) [13] (x86 version 2.4.6) from IBM Research. We have conducted our experiments on a dedicated 4-processor box (Intel Pentium 3(Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9) with hyper-threading enabled. We report results for 1, 2, 4, and 8 processors – 8 enabled virtually via hyperthreading. We execute each experiment 10 times and present performance data for the best-performing.

For each set of experiments, we report results for two JVM configurations. The first uses a fast, non-optimizing compiler (BaseVM) and the second employs an adaptively optimizing compiler [2] (PAOptVM). With PAOptVM, we employ pseudo-adaptation (PA) [4], to reduce non-determinism in our experimentation. We include results for both JVM configurations to show the performance impact of DBLFu-



(a) BaseVM



(b) PAOptVM

Figure 2. Efficacy of the LazyFuture implementation.

tures for systems that dynamically produce very different code quality.

The benchmarks that we investigate are from the benchmark suite in the Satin system [26]; we list them in Table 1 sorted by the rate of future generation. Each implements varying degrees of fine-grained parallelism. At one extreme is *Fib* which computes very little but creates a very large number of potentially concurrent methods. At the other extreme is *FFT* and *Raytracer* which implement few potentially concurrent methods, each with large computation granularity. Column 2 in the table is the input size that we employ and column three is the total number of potential futures in each benchmark.

In the final two columns of the table, we show the execution time in seconds for the serial version of the J5Future implementation. For this version, we invoke the `call()` method of each `Callable` object directly instead of submitting it to an Executor for execution.

5. Results

In this section, we first empirically compare the performance and scalability of Java 5.0 futures (J5Futures) and Lazy Futures (LazyFutures). We then evaluate the efficacy of DBLFutures and investigate its overhead.

5.1 Java 5.0 vs Lazy Futures

We first investigate the performance of the LazyFuture implementation in our system and compare it to that of J5Futures. For the J5Future implementation, we employ *Executor-Callable-Future* model for all potentially concurrent methods in each benchmark. We investigate two simple, but typical, Executors for this model: one that spawns a thread for every future submitted and another that uses a variable-length thread pool to execute futures. We refer to these implementations as *J5Future-ST* and *J5Future-TP*, respectively.

Since our benchmarks have a large number of fine-grained futures, many more than these two executors can handle, we identify spawning thresholds of computation granularity that enable the best performance experimentally for each system configuration and benchmark. We then parameterize the Executors with these thresholds. The modified Executors evaluate the submitted computation prior to spawning a new thread or employing the thread pool. If the granularity of the job is smaller than the corresponding threshold, the Executor simply invokes the `call()` method of the `Callable` object directly. Note that such hand-tuned Executors are not feasible in practice since optimal spawning thresholds vary significantly across applications, inputs, and execution environments, and doing so introduces a tremendous burden on the programmer. Nevertheless, we use these two hand-tuned Executors to represent cases where good spawning decisions are made given perfect knowledge of the underlying system and without incurring overhead for dynamic decision making and future splitting.

We present the average speedup over serial execution, across benchmarks, for parallel execution using J5Future-ST, J5Future-TP and LazyFuture respectively in Figure 2. For the LazyFuture implementation, we modify the benchmarks to eliminate the Executor as we describe in Sections 2 and 3. Graph (a) shows the results for BaseVM; graph (b) shows the results for PAOptVM; the x-axis in both is the number of processors that we use in each experiment. Note that for one processor, there is no spawning in all three approaches. Thus, the speedup values for one processor indicate the overhead of running Executors (for the two J5Future implementations) or the future splitting system (for LazyFutures).

The results with more than one processor show that the lazy future splitting system produces comparable performance to the hand-tuned Executors. In some cases, the LazyFuture system outperforms the Executors since it adaptively identifies tasks to split and spawn. J5Futures require that the user specify a static, fixed threshold to determine whether or not to spawn a concurrent task; the LazyFuture implementations does so automatically and adaptively according to underlying resource availability and program performance.

The absolute speedup values indicate however, that neither approach scales well in either JVM configuration. This is due to the significant overhead introduced by interface-based future implementations in Java. Such implementations create encapsulating objects regardless of whether the computation is inlined or executed in parallel, and thus, cause significant memory management overhead, which severely limits performance and scalability. We next evaluate the efficacy with which directive-based lazy futures address this limitation.

5.2 Directive-based Lazy Futures

We first compare the scalability of DBLFutures and LazyFutures. Both implementations share our splitting and spawning virtual machine infrastructure; the DBLFuture system however, employs the extensions that we describe in Section 3.

Table 2 shows the speedup of DBLFutures over LazyFutures for each benchmark. Columns 2-5 present results for increasing processor counts; Table (a) shows the results for BaseVM and (b) shows the results for PAOptVM. DBLFutures enable significant performance gains over LazyFutures for all configurations and processor counts. On average, the DBLFuture implementation is 9.1 to 10.8 times faster than LazyFutures for all experiments for the BaseVM and 1.8 to 4.4 times faster for the PAOptVM case. Moreover, the performance gains increase with the number of futures (e.g. Fib versus Raytracer). Since Fib is an extreme case relative to the other benchmarks, we also show the average speedups across benchmarks not including Fib. This average is 2.4 to 2.8 times faster for the BaseVM and 1.3 to 1.6 times faster for the PAOptVM case.

The primary reason for the performance improvement is the programming model. For LazyFutures, the JVM has the flexibility to decide whether to inline or spawn a future, but must always create the `Callable` and `Future` object due to its interface-based model. The DBLFuture employs a function-call based model, which (1) avoids the creation of `Callable` objects completely; (2) grants the JVM the flexibility to create a `Future` object only when it decides to spawn a future based on underlying resource availability and dynamic program behavior. Our in depth analysis of the performance gains shows that the benefits that DBLFutures achieve is due primarily to the avoidance of memory allocation and management.

Bench-	Processor Numbers			
marks	1	2	4	8
FFT	1.11 x	1.13 x	1.12 x	1.03 x
Raytracer	1.01 x	1.02 x	1.01 x	1.00 x
AdapInt	2.90 x	2.97 x	3.02 x	4.61 x
QuickSort	5.53 x	5.29 x	5.22 x	5.65 x
Knapsack	1.57 x	1.58 x	1.64 x	1.64 x
Fib	42.55 x	44.63 x	46.67 x	51.09 x
Avg	9.11 x	9.44 x	9.78 x	10.84 x
Avg(w/o Fib)	2.42 x	2.40 x	2.40 x	2.79 x

(a) BaseVM

Bench-	Processor Numbers			
marks	1	2	4	8
FFT	1.08 x	1.12 x	1.01 x	1.00 x
Raytracer	1.01 x	1.01 x	1.00 x	1.01 x
AdapInt	1.23 x	1.18 x	1.26 x	1.47 x
QuickSort	1.87 x	2.10 x	2.27 x	2.72 x
Knapsack	1.31 x	1.57 x	1.76 x	1.86 x
Fib	4.46 x	6.64 x	12.42 x	18.17 x
Avg	1.83 x	2.27 x	3.29 x	4.37 x
Avg(w/o Fib)	1.30 x	1.40 x	1.46 x	1.61 x

(b) PAOptVM

Table 2. Speedup of DBLFutures over Lazy Futures.

The improvements for PAOptVM are smaller than for BaseVM due to the efficient runtime services and dynamic code generation that PAOptVM performs (including aggressive optimization of object allocation). In addition, the performance difference between BaseVM and PAOptVM speedups increase with the number of processors. This is because the more processors that are available, the more acute the competition for system resources and services. Thus, by eliminating most of the unnecessary object allocation, DBLFuture is able to reduce the conflicts in parallel memory management, which provides additional performance gains.

We next analyze the overhead and scalability of our DBLFuture system in Table 3. The table contains one section each for the BaseVM (a) and the PAOptVM (b) configurations. We use T_i to represent the execution time of programs written using DBLFuture with i processors, and T_s for the execution time of the cor-

Benchmarks	T_s	T_s/T_1	T_1/T_2	T_1/T_4	T_1/T_8
FFT	41.61 s	1.00 x	1.88 x	3.09 x	2.86 x
Raytracer	162.40 s	1.00 x	1.93 x	3.66 x	3.78 x
AdapInt	43.04 s	0.97 x	1.98 x	3.85 x	6.19 x
Quicksort	14.65 s	0.91 x	1.83 x	3.28 x	3.87 x
Knapsack	88.36 s	0.97 x	1.86 x	3.68 x	3.43 x
Fib	8.17 s	0.31 x	1.99 x	3.96 x	4.26 x
Avg	59.70 s	0.86 x	1.91 x	3.59 x	4.07 x
Avg(w/o Fib)	70.01 s	0.97 x	1.90 x	3.51 x	4.03 x

(a) BaseVM

Benchmarks	T_s	T_s/T_1	T_1/T_2	T_1/T_4	T_1/T_8
FFT	7.55 s	0.99 x	1.60 x	1.99 x	1.88 x
Raytracer	19.81 s	0.99 x	1.90 x	3.22 x	3.84 x
AdapInt	24.34 s	0.93 x	1.73 x	3.43 x	5.24 x
Quicksort	6.14 s	0.88 x	1.90 x	3.01 x	3.44 x
Knapsack	10.70 s	0.96 x	1.84 x	2.76 x	2.58 x
Fib	5.59 s	0.34 x	1.98 x	3.94 x	4.02 x
Avg	12.36 s	0.85 x	1.83 x	3.06 x	3.50 x
Avg(w/o Fib)	13.71 s	0.95 x	1.79 x	2.88 x	3.40 x

(b) PAOptVM

Table 3. Overhead and scalability of directive-based lazy futures

responding serial version, which is listed (in seconds) in Columns 2 and 7 of Table 3. Note that due to its function-call based coding style, this serial version is much faster than the serial version we used as the baseline for J5Futures and LazyFutures (see Column 4 and 5 of Table 1). Therefore, we are setting a higher standard here to evaluate our DBLFuture system against, and the speedup values in Table 3 have different scales from those in Figure 2.

Columns 3 and 8 show the T_s/T_1 value, our overhead metric. Since there is only function call overhead for each potential future invocation in the serial version, the difference between T_1 (single processor) and T_s reveals three sources of overhead: (1) the bookkeeping employed to maintain the shadow future stack, (2) the activities of the future profiler, controller, and compiler, and (3) the conditional processing required by the DBLFuture version for the storing and first use of the value returned by a potential future call. The JVMs perform no splitting in either case. This data shows that our DBLFuture implementation is very efficient: only negligible overhead is introduced for most benchmarks. The worst case is *Fib*, which shows a 3x slowdown. This is because the Fib benchmark performs almost no computation for each future invocation (computing a Fibonacci value). The results for this benchmark represents an upper bound on the overhead of our system. The C implementation for a similar parallel system, called Cilk, introduce a similar overhead for this benchmark (3.63x slowdown [9]). Our system however, significantly outperforms the Java version of Cilk (JCilk) which imposes a 27.5x slowdown for this benchmark [8]).

The remaining columns for each JVM configuration show the speedups gained by DBLFuture when we introduce additional processors (which we compute as T_1/T_i as we increase i, the processor count). For the BaseVM case, the execution time on N processors scales almost to $1/N$ (average speedup is 1.91x, 3.59x, 4.07x for processor 2, 4, 8 respectively), that is, our system enables approximately linear speedup for most of the benchmarks that we investigate. Note that our hardware has 4 physical processors and uses hyperthreading to emulate 8 processors. Despite improvements in code quality enabled by the PAOptVM case, the DBLFuture version is able to extract average performance gains of 1.83x, 3.06x, 3.50x for 2, 4, and 8 processors, respectively. Again, we list the average data excluding Fib in the last row of the table to avoid Fib skewing the results. In summary, our DBLFuture implementation achieves scalable performance improvements with negligible overhead.

6. Related Work

Language and runtime support for fine-grained futures has been studied widely for functional languages [17, 19] and C++ [27]. *Lazy task creation* (LTC) [19], in particular, has inspired many system designers interested in exploiting fine-grained parallelism [21, 10, 9, 24] in a lazy fashion: execute sequentially first, and in parallel if necessary.

Lazy Java Futures (LazyFutures herein) [29] is the first Java Virtual Machine implementation in support of fine-grained futures. It exploits both general thread scheduling and performance sampling in the JVM to guide splitting and spawning of futures lazily based on underlying resource availability and dynamic computation granularity. In all prior systems, spawning is triggered only by a blocked task or idle processor.

Our *DBLFuture* system extends LazyFutures to support a directive-based programming model for using futures in Java (as opposed to an interface-based, object-oriented approach of LazyFutures and all other approaches to Java futures). As a result, DBLFutures simplify the introduction of parallelism into programs and enables greater flexibility to the JVM to implement potentially concurrent code regions as efficiently as possible.

There are many previous works that support parallel programming linguistically, either language-based, i.e., through the addition of new keywords in the language (e.g., Cilk [5], JCilk [18], X10 [7], Fortress [1]), or directive-based (e.g. OpenMP [20]). Many programming languages support the future construct to some extent, either via a library interface (e.g., Java [14], C++ [27]), or directly (e.g., Multilisp [22], C [6], X10 [7], Fortress [1]). Some concurrent logic programming languages (e.g., OZ [23]) generalize the concept of futures to rather extremes. In such languages, all logic variables are conceptually future variables: they can be bound by a separate thread and threads that access an unbound logic variable will be blocked until a value is bound to this variable. We follow the directive-based approach instead of language-based approach for easy implementation. The focus of our paper, however, is not the linguistic programming model itself, instead, we are interested in the performance impact of different future implementations for Java. We find that a linguistic approach provides the JVM and compiler with more flexibility to interpret future calls efficiently.

New extensions to the Java language can also be implemented by transforming the new constructs to calls to runtime libraries via either source-to-source transformation [8, 12] or bytecode rewriting [3, 16]. This approach has the advantage of portability and easy implementation since it does not require JVM modification. We show, in this work, however, that JVM support in a way that takes advantage of extant JVM services is important to achieve high performance and scalability. Also, our experiences show that by leveraging extant JVM design and implementations, and by eliminating extra abstraction layers, such JVM support to new language constructs can be feasible and sometime even simpler to implement comparing to higher-level alternatives.

The authors in [28] propose *safe* futures for Java. Their system uses object versioning and task revocation to enforce the semantic transparency of futures automatically so that programmers are freed from reasoning about the side-effects of future executions and ensuring correctness. Safe futures focus on automatic assurance of safe concurrent execution of futures, while our system's foci are the easy programming and performance aspects of futures. Therefore, the two are complementary and we plan to investigate their integration in our system as part of future work.

7. Conclusions

We investigate the implementation of the future parallel programming construct for Java. Futures provide a simple and elegant way for programmers to introduce concurrency into their programs. In the current Java future APIs [14], programmers add futures to their applications using the *Executor-Callable-Future* programming model. We evaluate this model for programs with fine-grained parallelism, and find that it introduces significant performance overhead. Moreover, this programming methodology imposes an undue burden on the average users. To address these limitations, we introduce directive-based lazy futures (DBLFutures) and present its necessary Java Virtual Machine compiler and runtime extensions.

With DBLFutures, programmers can easily introduce parallelism into their programs by annotating local variables that receive results from function calls that can be safely executed in parallel. Our DBLFuture system builds upon and extends a lazy future implementation to enable the runtime to decide when to split and spawn annotated future calls according to underlying resource availability and dynamically determined computation granularity. In addition, our DBLFuture implementation avoids unnecessary object creation that causes significant memory management overhead in prior approaches of futures in Java. We empirically evaluate our future implementation using different Java Virtual Machine configurations and common Java benchmarks that implement fine-grained parallelism. Our results show that our DBLFuture system imposes negligible overhead on serial executions and is significantly more scalable than all prior implementations of futures in Java.

Acknowledgments

We thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by Microsoft and NSF grants CCF-0444412 and CNS-0546737.

References

[1] E. Allan, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 0.785. Technical report, Sun Microsystems, 2005.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[3] The AspectJ Project. `http://www.eclipse.org/aspectj/`.

[4] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *International symposium on Memory management*, pages 143–151, 2004.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[6] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 95–113, London, UK, UK, 1990. Pitman Publishing.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM conference on Object oriented programming systems languages and applications*, pages 519–538, 2005.

[8] J. S. Danaher. The jcilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM conference on Programming language design and implementation*, pages 212–223, 1998.

[10] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996.

[11] J. Henry C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM Press.

[12] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Wrkshop on Memory system performance and correctness*, pages 82–91, New York, NY, USA, 2006. ACM Press.

[13] IBM Jikes Research Virtual Machine (RVM). `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[14] JSR166: Concurrency utilities. `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency`.

[15] JSR 175: A Metadata Facility for the JavaTM Programming Language. `http://jcp.org/en/jsr/detail?id=175`.

[16] M. Karaorman and P. Abercrombie. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Form. Methods Syst. Des.*, 27(3):275–312, 2005.

[17] D. A. Kranz, J. R. H. Halstead, and E. Mohr. Mul-T: a high-performance parallel Lisp. In *ACM Conference on Programming language design and implementation*, pages 81–90, 1989.

[18] I.-T. A. Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.

[19] E. Mohr, D. A. Kranz, and J. R. H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991.

[20] OpenMP specifications. `http://www.openmp.org/specs`.

[21] J. Plevyak, V. Karamcheti, X. Zhang, and A. A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *ACM/IEEE conference on Supercomputing*, page 41, 1995.

[22] J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[23] S.Haridi and N.Franz. Tutorial of Oz, Mozart documentations. `http://www.mozart-oz.org/documentation/tutorial/~index.html`.

[24] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *ACM symposium on Principles and practice of parallel programming*, pages 60–71, 1999.

[25] K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages. In *ACM conference on Programming language design and implementation*, pages 320–333, 1997.

[26] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[27] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *ACM symposium on Principles and practice of parallel programming*, pages 208–217, 1993.

[28] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *ACM conference on Object oriented programming systems languages and applications*, pages 439–453, 2005.

[29] L. Zhang, C. Krintz, and S. Soman. Efficient Support of Fine-grained Futures in Java. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2006.