EFFICIENT SUPPORT OF FINE-GRAINED FUTURES IN JAVA

Lingli Zhang Chandra Krintz Sunil Soman
Computer Science Department
University of California, Santa Barbara
{lingli_z,ckrintz,sunils}@cs.ucsb.edu

ABSTRACT
A future is a parallel programming language construct that enables programmers to specify potentially asynchronous computations. We present and empirically evaluate a novel implementation of futures for Java. Our futures implementation is a JVM extension that couples estimates of future computational granularity with underlying resource availability to enable automatic and adaptive decisions of when to spawn futures in parallel or to execute them sequentially. Our system builds from, combines, and extends (i) lazy task creation and (ii) a JVM sampling infrastructure previously used solely for dynamic and adaptive compilation. We empirically evaluate our system using different benchmarks, triggers for automatic spawning of futures, processor availability, and JVM configurations. We show that our future implementation for Java is efficient and scalable for fine-grained Java futures without requiring programmer intervention.

KEY WORDS
Java, futures, fine-grained, parallel programming, task scheduling, profile-guided

1 Introduction

As multi-processor computer systems become ubiquitous, it is becoming increasingly important to effectively support parallel programming constructs that are easy to use in popular, high-level languages. One such construct is the *future*. A future identifies a potentially asynchronous computation that can be executed in parallel. The construct was first introduced in Multilisp [17], and has been adopted by many languages including Java J2SE 5.0 [11] and X10 [3]. The original rationale behind futures is that "the programmer takes on the burden of identifying *what* can be computed safely in parallel, leaving the decision of exactly *how* the division will take place to the run-time system" [14].

Using this model, a programmer specifies *all* code regions that *can* execute in parallel, and the runtime decides when to do so. For programs that contain fine-grained, independent computations, it is possible for a programmer to specify a very large number of futures. It is, therefore, vital for performance that the runtime implementation of futures be efficient, and effectively amortize the cost of spawning a future in parallel, or execute the future sequentially (inline it into the current context). Naïve future implementations (e.g. one thread per future or with thread pool support) can result in significant overhead, and inefficient, even degraded, execution. Such future implementations in Java can quickly bring the system to a halt due to the multiple layers of abstraction and virtualization in the Java Virtual Machine (JVM) for the support of system services, such as, threads, memory management, and compilation.

To limit the number of independent contexts that are spawned for fine-grained futures, programmers commonly specify thresholds to identify futures that will perform enough computation to warrant parallelization. This approach is time-consuming, and error prone. The thresholds are specific to, and different across applications, inputs for the same application, available underlying hardware resources, and execution environment, thereby, requiring significant effort and expertise by the programmer to identify optimal, or even efficient settings. Moreover, the requirement that users participate in deciding which futures to spawn or inline, is inconsistent with the original design goal of futures of placing a minimal burden on the programmer. In this paper, we investigate a runtime implementation that efficiently supports fine-grained futures without requiring programmer intervention with parallelization decisions.

Prior work proposes several solutions for such support within functional languages or C++ [13, 14, 23]. In this paper, we focus on supporting efficient fine-grained futures in Java. In current Java future APIs [11], futures must be submitted to a user-defined `Executor` for execution. How a future is dispatched depends on the implementation of the `Executor`. The decision of whether to spawn or inline futures depends on many factors including the execution behavior of the future, underlying resource availability, as well as the capability of the execution environment. Unfortunately, the library-based `Executor` model is not capable of obtaining all of the above information to make well-informed decisions.

In our work, we follow an alternative, runtime-based, approach and extend the JVM runtime to effectively support futures. We do so since the JVM has access to low-level information about the executing program, and underlying resource availability. Our approach, which we call *lazy futures*, builds from, combines, and extends (i) lazy task creation [14] and (ii) a JVM program sampling infrastructure (common to many state-of-the-art JVM implementations) previously used solely for dynamic and adaptive compiler optimization. We couple these techniques with dynamic state information from the underlying, shared-memory, multiprocessor resources, to adaptively identify

Bench-marks	Inputs size	Total# of futures	CPU 1.60GHz proc#=2(base)	CPU 1.60GHz proc#=4(base)	CPU 2.40GHz proc#=2(base)	CPU 2.40GHz proc#=2(opt)
AdapInt	0-250000	5782389	7000000	8000000	17000000	19000000
FFT	2^{18}	262143	4096	32768	16384	65536
Fib	38	39088168	30	32	36	33
Knapsack	24	8466646	5	7	6	4
Quicksort	2^{24}	8384315	131072	131072	131072	524288
Raytracer	pics/balls.nff	265409	32	16	32	64

Table 1. Evidence that threshold values vary widely across configurations for the same program and input. We identified these thresholds empirically from a wide range of threshold values.

when to spawn or inline futures.

We empirically evaluate our system using a number of Java programs, implementation approaches, and runtime system configurations. We investigate mechanisms that trigger automatic spawning of futures that consider estimates of future granularity, processor availability, and a hybrid of both. Our results show that we are able to implement Java futures in a way that requires no programmer intervention into the spawning decisions of futures, and that is scalable and efficient given the available resources and virtualized execution environment of the JVM. Moreover, our system enables efficient parallel execution of both coarse and fine grain futures; for fine-grained futures our system enables performance that is similar to hand-tuned, threshold-based alternatives.

We organize the rest of the paper as follows. Section 2 describes the design and implementation details of our lazy futures system. We then empirically compare the various implementation alternatives and evaluate the overall efficacy of our system in Sections 3 and 4. The remainder of the paper includes related work (Section 5), and our conclusions (Section 6).

2 System Design and Implementation

Lazy futures is a futures implementation for Java that we propose to support efficient execution of fine-grained futures. Our goal is to eliminate the need for programmers to decide when, and how to spawn futures in parallel for applications with fine-grained futures. For such applications, programmers commonly specify a computational granularity that amortizes the cost of spawning a future in parallel.

In practice, this threshold is difficult and tedious to identify, and can have a large impact on performance since the optimal values vary significantly across applications, inputs, available underlying hardware resources, and execution environments. To validate this claim, we empirically identified the thresholds for optimal performance for six benchmarks. We present these thresholds in Table 1. We gathered results on two machines: one with four 1.60GHZ processors, the other with two 2.40GHZ processors. On the 4-processor machine, we collected data with 2 as well as 4 processors. On the 2-processor machine, we used two different configurations of the same JVM. We provide specific details of our methodology in Section 3. This data confirms that the best thresholds vary across different configurations. Lazy futures free the programmers from the task of threshold specification, and enable the system to decide when and how to spawn futures in parallel adaptively.

Our implementation of lazy futures is inspired by the technique proposed by Mohr et al. [14], called *lazy task creation* (LTC). LTC initially implements all futures as function calls. The system then maintains special data structures for the computation of future's parent, the caller (called a continuation), to be spawned. When there is an idle processor available, the idle processor steals continuations from the first processor and executes code in parallel with the future. Similar techniques are employed in many systems to support fine-grained parallelism [16, 6, 7, 21].

Our system, although similar, is different from these prior approaches in two ways. First, we combine information about computation granularity with resource availability. Prior work commonly considers only the latter, since estimating the computation granularity at runtime is complex, and can introduce significant overhead. Our implementation is, however, targeted at state-of-the-art JVMs, which implement a low-overhead runtime profiling system that the runtime uses to guide adaptive compilation and optimization [1, 15, 19, 10]. We exploit this mechanism to estimate the computational granularity of futures.

The second unique aspect of our implementation is that we do not employ a worker-based, specialized runtime system for futures. Systems like LTC typically associate a worker with each physical processor, and this worker is responsible for executing the current task, stealing tasks from other workers, and managing the task queues. Such systems assume that futures (or special kind of tasks that the system supports) are the only kind of parallel activity in the system. In addition, these systems map runtime threads directly to operating system (OS) threads. Such a setup is not appropriate for a JVM since this would equate to mapping worker threads to Java threads (which are themselves mapped to OS threads), thereby, adding an additional level of indirection, and overhead to scheduling. Moreover, a JVM would need to accommodate varied types of parallel constructs specified in Java, other than futures.

In our system, we integrate future management with the existing thread scheduling mechanism in the JVM. When the system identifies a future to spawn on the runtime call stack of a thread, the system splits the thread into two – one that executes the future,and the other that performs

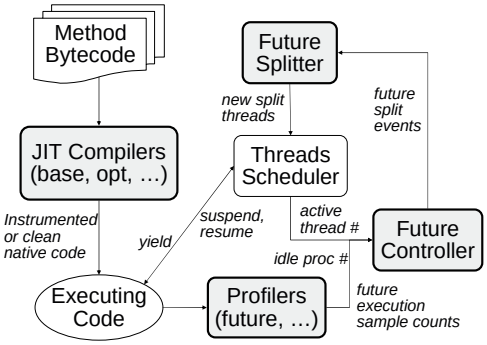


Figure 1. Overview of lazy future implementation.

the continuation. Both threads are considered Java threads by the thread scheduler. With this implementation, we take advantage of the highly-tuned JVM thread scheduler, synchronization, and load-balancing mechanisms, which significantly simplify the implementation of futures.

2.1 Implementation overview

Figure 1 overviews our system. All shaded components identify our extensions to the JVM. After a class is loaded by the class loader, the method bytecodes are translated to native code by the Just-in-time (JIT) compiler (non-optimizing, as well as optimizing). The compiler may insert instrumentation into the native code to collect profiling information from the program that the compiler can later use to perform optimizations.

We extend the JIT compilers to insert a small stub at the entry point and exit point of every future call. Initially, our system treats every future call as a function call, i.e., the system executes the code on the stack of the current thread. At the same time, we maintain a small side stack for each thread, called a *future stack* (See Figure 2). Every entry in the future stack has two words, one is the offset of a future frame on the current stack, the other is the sample count that holds an estimate of how long the future call has executed. The stubs push an entry onto the future stack at the beginning of a future call, and pop the entry when exiting the future call. We implemented these stubs carefully in the JIT compilers, and ensure that they are always inlined, to avoid unnecessary overhead.

To estimate the computation granularity of futures, we extend the existing JVM sampling system. In our prototype JVM, light-weight method sampling occurs at every thread switch (approximately every 10 ms), which increments sample counts of the top two methods on the current stack. Methods with sample counts exceeding a certain threshold will be identified as hot methods, and recompiled with higher levels of optimizations. We extend this mechanism by also incrementing the sample counts of executing futures. These sample counts provide our system with an estimate of how long the futures have executed. Our scheduling system spawns futures whose sample counts exceed a particular threshold. This process avoids spawning short-running futures – the overhead of which cannot be amortized by the benefits from parallel execution.

The system feeds the future sample counts into the *future controller*, which couples the sample counts with dynamic system resource information from the thread scheduler, e.g., the number of currently active threads and idle processors, to adaptively make decisions about splitting futures, in order to enable additional parallelism.

If the future controller decides that it is beneficial to split a future, it creates a *futureSplitEvent* that contains information about the future, such as the frame offset and sample counts. The controller forwards the event to the *future splitter*, which splits the current thread into a future thread and a continuation thread, and places both threads on the appropriate queue of the thread scheduler for further execution. Note that both the future controller and future splitter are services invoked by the current thread, when the thread yields to enable thread switching. Therefore, we require no additional synchronization since the system implements this process on a per-thread basis.

2.2 Future splitting triggers

Ideally, we should spawn a future when there is an idle processor. We refer to this approach *idleProc triggered*. In our system, future splitting is initiated by the running thread, and only occurs during thread switching. If a processor becomes idle during execution, there is a delay before a thread detects this and makes the splitting decision. There is also a small delay between when a future is spawned, and when it is scheduled to execute. Thus, the idleProc triggered policy may not utilize the system resource fully in some cases.

One alternative is to saturate the system with futures. To implement this policy, we maintain twice as many threads as processors for futures that the system selects. That is, if the sample count of a future call on stack exceeds the threshold, and the current number of active threads is less than twice the number of processors, the current thread will be split to make the future call a parallel call. We refer to this approach *sampleCount triggered*. This policy helps to pre-saturate the system if enough parallelism is available, but imposes a delay for "learning" that a future is long-running, i.e., the time it takes for the sample count to exceed the threshold.

Therefore, we consider a hybrid approach, which we call *sample+idle triggered*. Note that in all policies, since the system performs future splitting (spawning) only at thread switching, it automatically eliminates futures with granularity of less than 10 ms from spawning.

2.3 Future splitter

Figure 2 overviews our process for splitting futures. In the figure, the current thread has three future calls on its stack. At some point, the future controller decides that it is worthwhile to spawn the oldest future call with sample count 10 for parallel execution. The dark line identifies the split point on the stack. The future splitter then creates a new thread for the continuation of the spawned future call, copies the stack frames below the future frame, which cor-

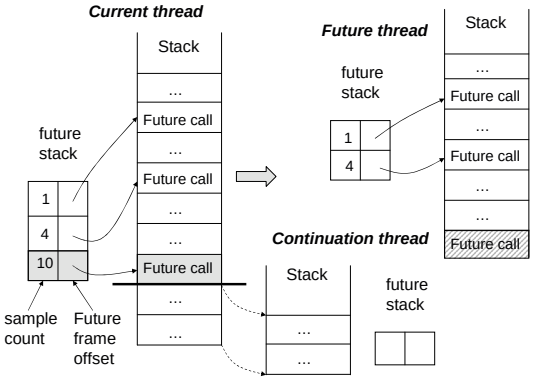


Figure 2. A future is lazily created.

responds to the continuation, restores the execution context from the stack frames, and resumes the continuation thread at the return address of the spawned future call. Note that we choose to create a new thread for the continuation instead of the spawned future, so that we do not need to setup the execution contexts for both threads. The spawned future call becomes the bottom frame of the current thread. The system deletes the future stack entry so that it is no longer treated as potential future call.

2.4 Optimizing synchronizations

If the result of a future is used by its parent, the system will check whether or not the result is available. If it is not, the parent blocks until the future completes. This synchronization process can be avoided if the future is not spawned. In this case, the result of the future is ready at the time the future call returns to its parent, and thus, will always be ready at its usage points. To optimize this case, we add a *onStack* flag to each future object. We initialize the flag to true and set it to false if the future splitter spawns the future. When the result of a future is requested, if its onStack flag is true, the system returns the result directly, otherwise, we synchronize the process with its future execution.

3 Experimental methodology

We implemented lazy futures in the open source Jikes Research Virtual Machine (JikesRVM) [9] (x86 version 2.4.2) from IBM Research. To evaluate the efficacy of our approach, we also implemented two other alternatives to support futures in Java: one that spawns a thread for every future and another that uses a variable-length thread pool to execute futures. We refer to these implementations as *singleThread* (ST), *thread Pool* (TP), respectively.

To investigate the impact of lazy futures on different application types, we developed two sets of benchmarks. The first set includes *Crypt*, *MonteCarlo*, *Series* and *SparseMatmult*, which is a subset of the multithreaded version of Java Grande Benchmark Suite [18]. These four benchmarks are chosen because there is no mutual dependency between spawned parallel tasks in these benchmarks, which makes them suitable to be expressed by futures. The structure of these benchmarks is similar: the main thread spawns several futures to compute subtasks, and then it waits for all futures to finish. The number of futures to spawn can be specified by users on the command-line option, and is usually set to the number of processors available. This kind of applications represents coarse-grained parallelism. The singleThread implementation is usually sufficient to handle such applications. We use this set of benchmarks to evaluate the overhead introduced by our lazy future implementation.

The second set of benchmarks includes *AdapInt, FFT, Fib, Knapsack, QuickSort, Raytracer*. All of the programs employ a divide and conquer model. We adopt them from the examples provided by the Satin system [22]. The recursive nature of these benchmarks results in excessive number of futures with very different granularities. We use this set of benchmarks to evaluate whether our lazy future implementation can make effective future splitting decisions automatically and adaptively.

We conduct 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. Thus, we report results for up to 8 processors. We execute all benchmarks repeatedly and present the minimum. For each set of experiments, we report results for two JVM configurations respectively: one with the non-optimizing (baseline) compiler and the other with the highly-optimizing (opt) compiler. For the optimizing configuration, we use the adaptive setting [1] which optimizes frequently executed methods only. To eliminate non-determinism, we use the pseudo-adaptive configuration [2], which mimics the adaptive compiler in a deterministic manner by applying the optimizing compiler to code according to an advice file that we generate offline. We include results for both JVM configurations to show how well our future implementation identifies long-running futures. Unoptimized futures will execute for a longer duration than the optimized versions, and consequently, our system will automatically adapt to the code performance and execution environment, and make different spawning decisions.

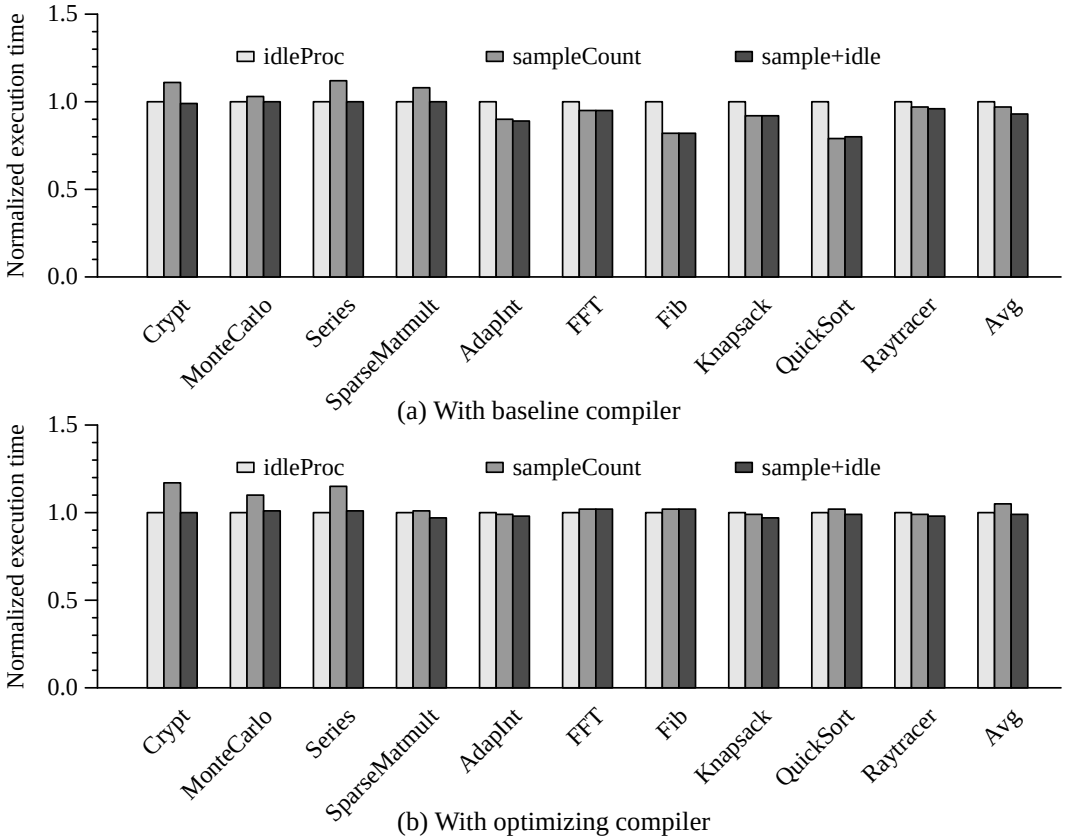
Finally, we use a sample count of 5 as the splitting threshold for the sampleCount policy in our results. We selected this value empirically from a wide range of values that we experimented with. We found that this value, across benchmarks, imposed only a small "learning" delay, and effectively identifies futures for which the overhead of spawning is amortized by parallel execution.

4 Results

In this section, we first evaluate the efficacy of different future splitting triggers. Then we analyze the performance impact of our lazy future implementation in detail for all benchmark sets.

4.1 Comparison of Splitting Triggers

As we discussed in section 2.2, in our system, future splitting can be triggered by either available idle processors or



(a) With baseline compiler

(b) With optimizing compiler

Figure 3. Comparison of future splitting triggers.

high future sample count, or both. In this section, we compare performance of all three triggers.

Figure 3 shows the execution time for all benchmarks with different splitting triggers. We normalize the data relative to *idleProc* for easy comparison. The first four benchmarks are from the JavaGrande suite, and the rest are from our divide and conquer suite. Graph (a) shows the results when we use the baseline compiler and graph (b) shows results with the pseudo adaptive optimization setup.

The data indicates that for applications with few coarse-grained futures (the first four benchmarks), the *sampleCount* triggered policy is less effective than the *idleProc* policy. This is due to the delay required to "learn" whether a future will be short or long running by the *sampleCount* policy – when there are several idle processors available.

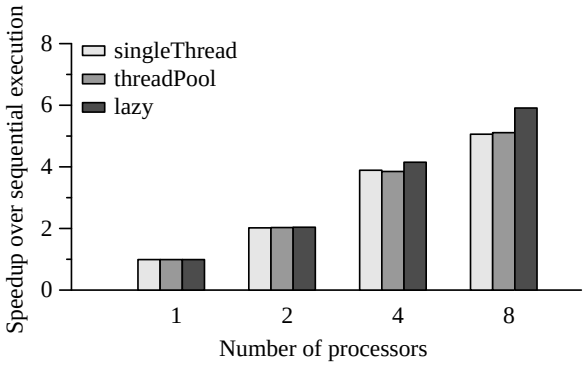
For applications with a large number of fine-grained futures (the remaining benchmarks), the *sampleCount* trigger outperforms the *idleProc* trigger in most cases since it helps saturate the system with qualified futures to utilize the system better. This trend is more apparent when the baseline compiler is used. This is because the baseline compiler produces unoptimized code for both the system and the application, which makes the process of detecting idle processors, splitting and scheduling futures take longer. Thus, pre-saturating the system using the sampleCount trigger makes a bigger difference. In summary, by combining both triggers, the hybrid *sample+idle* policy achieves the best performance among all triggers. All results in further sections use the hybrid trigger.

4.2 JavaGrande Performance

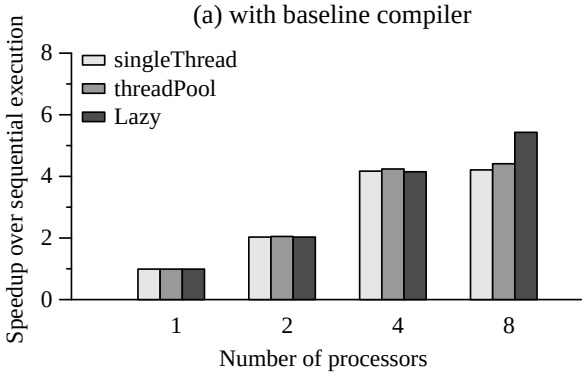
In this section, we evaluate the performance impact of our lazy future implementation on the four JavaGrande benchmarks. This set of benchmarks represents applications with a small number of coarse-grained futures.

Figure 4 shows the average speedup over the sequential version of each benchmark. The x-axis is the number of processors used. Note that the 8-processor case is actually the 4-processor case with hyper-threading. We set the number of futures in the applications to the number of processors used. We present three implementation alternatives: one thread per future (*singleThread*), variable-length thread pool (*threadPool*), and our lazy future implementation (*lazy*). Graph (a) and (b) are results for the baseline compiler and the optimizing compiler, respectively.

The data shows that with baseline compiler, all three implementations produce similar average performance: 1% overhead with one processor and around 2x speedup with two processors. When there are more processors available and more futures created, the threadPool implementation starts show a small improvement over the singleThread implementation. Our lazy futures implementation is competitive with the other alternatives, and outperforms them on average as the processor count increases. Note that lazy futures require a "learning time" of at least 10ms (time for one thread switching) for each future spawned to decide if the computation time warrants parallelization. The other two alternatives do not require a learning time.

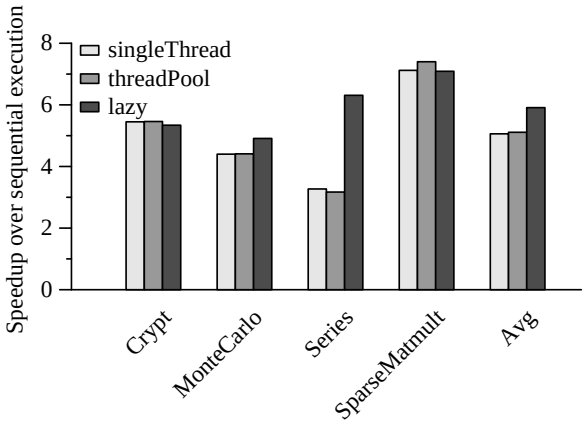


(a) with baseline compiler

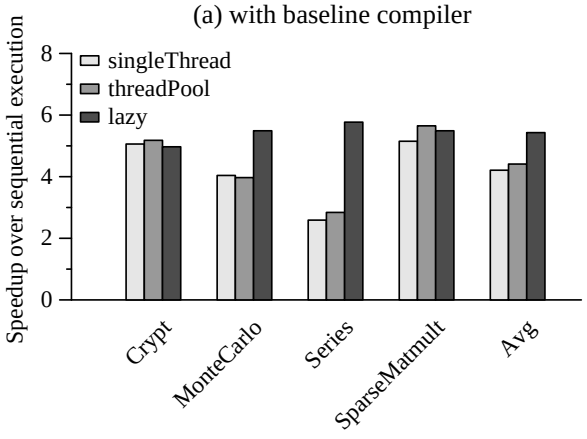


(b) with optimizing compiler

Figure 4. Average speedups: JavaGrande benchmarks.



(a) with baseline compiler



(b) with optimizing compiler

Figure 5. Individual JavaGrande benchmark speedups when we employ 8 processors.

To investigate these results in greater detail, we present speedup of the individual benchmark in Figure 5 for the 8 processor data. Graph (a) and (b) are the results with the baseline compiler and the optimizing compiler respectively. This figure shows that the lazy future implementation does introduce some overhead ($< 2\%$) for two benchmarks (*Crypt*, *SparseMatmult*) due to the learning delay. However, for the other two benchmarks, especially *Series*, this slight splitting delay actually improves performance significantly. We believe that in this case, the slight slowdown of future creations of our system reduces the contentions of system resources, such as cache conflicts, comparing to the other alternatives.

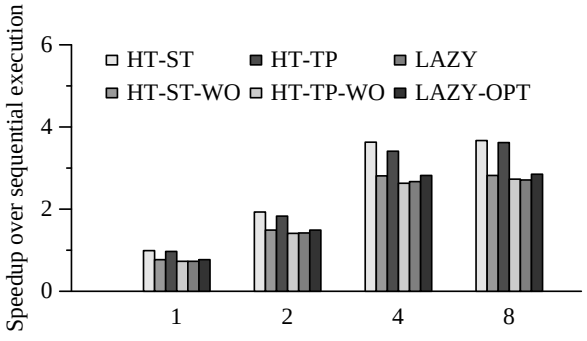
The average speedup with 8 processors is 5.0x for *singleThread*, 5.1x for *threadPool*, 5.9x for *lazy* when the baseline compiler is used. With the optimizing compiler, the average speedup is 4.2x for *singleThread*, 4.4x for *threadPool*, and 5.4x for *lazy*. In both configurations, our lazy future system outperforms the other two on average.

4.3 Divide and Conquer Performance

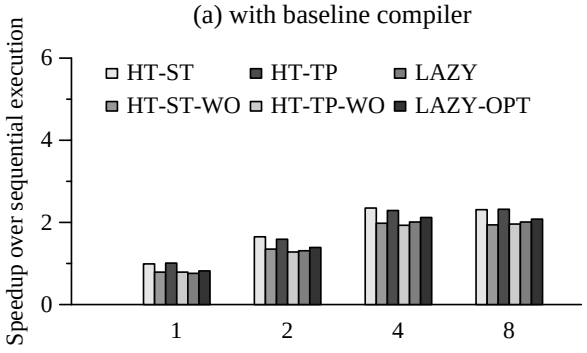
We next evaluate the performance impact of our lazy future implementation for the divide and conquer benchmark suite. We compare our approach to the singleThread and threadPool alternatives above using a hand-tuned granularity threshold. We identify the best performing thresholds experimentally for our various configurations and benchmarks. These two alternatives represent the case where the programmer specifies the threshold for spawning given perfect knowledge of the underlying system. This in practice is not feasible for all inputs, operating and runtime systems, and processor configurations, and it introduces a tremendous burden on the programmer. Our lazy future system requires only that the programmer specify which code regions can execute in parallel. Comparing our lazy future system to the singleThread and threadPool with the best, hand-tuned thresholds indicates the degree to which our system makes the appropriate spawning decisions.

We consider an additional configuration in our result set for these benchmarks. Using the current Java Concurrency Utilities [11], the system will create a future object for each future regardless of whether it is executed inlined or in parallel. In the hand-tuned alternatives, we do not create future objects if the future computational granularity is below the threshold. To investigate and report the overhead of this object allocation and to show the overhead inherent in doing so, we also include configurations of the hand-tuned alternatives that create future objects for *all* future instances even those that are below the threshold; however, we only spawn those above the threshold.

Figure 6 shows the average speedup over the sequential version for our divide and conquer benchmarks (fine-grain parallelism). The x-axis is the number of processors that we used for each experiment. The first two bars are results for the singleThread (ST) implementation with hand-tuned (HT-) thresholds, the middle two bars are results for the threadPool (TP) implementation with hand-

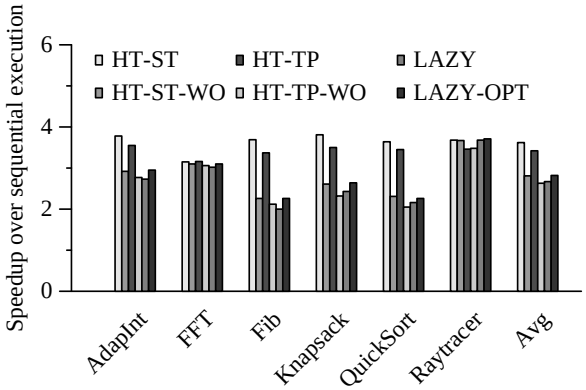


(a) with baseline compiler

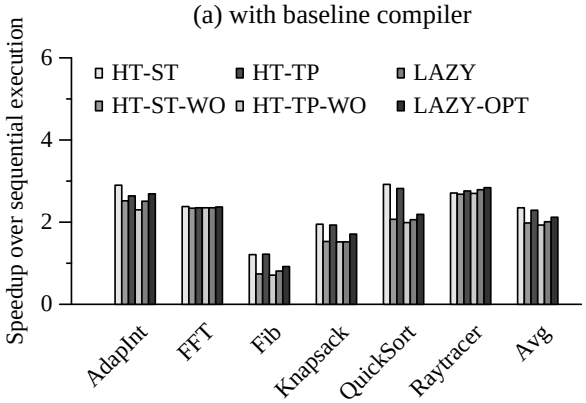


(b) with optimizing compiler

Figure 6. Average speedups: Divide and conquer benchmarks.



(a) with baseline compiler



(b) with optimizing compiler

Figure 7. Individual divide and conquer benchmark speedups for 4 processors.

tuned (HT-) thresholds. The last two bars are results for the lazy (LAZY) implementation, without and with optimizing synchronizations (-OPT) (Section 2.4). We use "-WO" to identify the configurations that we create wrapper objects for all future instances for the hand-tuned alternatives.

The data indicates that the overall speedup for this benchmark set is less than that of the JavaGrande benchmarks due to the fine-grained nature of these programs. Our lazy future implementation produces comparable, in some case better, performance than the hand-tuned thresholds – when we exclude the overhead of object allocation (HT-ST-WO and HT-TP-WO). The better performance is due to the fact that thresholds specified by programmers are static, and thus do not adapt to resource availability as our lazy future implementation does.

The differences between HT-ST-WO and HT-ST, or HT-TP-WO and HT-TP show that the extra unnecessary object allocation has significant performance impact on applications with fine-grained futures, although an optimizing compiler reduces the differences to some degrees (see Figure 6(b)). These differences imply that there is a large potential performance gain for our lazy futures implementation if the system is able to avoid creating future objects for future calls executed inlined. To achieve this, we believe that the language constructs ([17, 3]) as opposed to library constructs (e.g., the Java Future API) will provide the JVM more flexibility and opportunities of optimizations, and thus, enable more efficient support of fine-grained futures. We plan to investigate this hypothesis in depth in our future work.

Since these benchmarks almost always saturate the system with a large number of futures, hyper-threading does not help to improve the performance. Therefore, we show individual speedups with 4 processors for this set of benchmarks in Figure 7 to enable a more detailed analysis. This figure shows as more futures are created (more than 5 million for four benchmarks, see the second column of Table 1), the larger is the difference between HT-ST and HT-ST-WO. The Fib benchmark represents the worst case by creating almost 40 million futures object. The optimizing compiler is able to reduce the overhead primarily by inlining object allocation and initialization; however, the overhead is still significant.

Finally, to show the frequency of future spawning, we present Table 2. The table lists the number of Java threads created by each implementation alternatives with 4 processors. Since the "-WO" configurations have same thread number as its corresponding non-WO version and LAZY and LAZY-OPT also have similar counts, we only show numbers for HT-ST, HT-TP, and LAZY. "base" stands for the baseline compiler, and "opt" stands for the optimizing compiler. Note that each configuration has different threshold, so the specific values are incomparable. Instead, the data shows the efficacy of our lazy future system by comparing the thread number created by the LAZY imple-

Bench-marks	HT-ST		HT-TP		LAZY	
	base	opt	base	opt	base	opt
AdapInt	227	230	70	62	52	42
FFT	18	29	14	26	43	36
Fib	31	29	158	17	66	50
Knapsack	137	44	77	144	105	29
Quicksort	150	77	103	55	103	76
Raytracer	266	29	30	20	100	48

Table 2. Number of Java threads spawned.

mentation to the number of futures created by these applications (see the second column of Table 1). In summary, our lazy future system is able to make intelligent future inlining/spawning decisions automatically and adaptively, based on dynamic information of system resource availability and future granularity.

5 Related Work

Load-based inlining [13] was the first approach proposed to address the fine-grained future problem. The idea is to make spawning decision at the creation time based on the system load. A future is computed parallelly if there is enough available resource. Otherwise, it is inlined. One major drawback of this approach is that the decision is not revocable: once a future is inlined, it cannot be parallelized anymore. Task starvation may occur due to imbalance work load and bursty task creation.

Lazy task creation (LTC) [14] is a more elaborate scheme to support fine-grained futures. In this approach, all futures are initially evaluated like a sequential call. But the system maintains minimum information to spawn the continuations of futures retroactively if a future is blocked, or a computation resource becomes available. This principle of sequential first, parallel retroactively if necessary, can be found in many systems that target fine-grained parallelism [16, 7, 6, 20], each with its own contexts and refinements. Our system follows the laziness principle as well. However, we believe that our system is the first effort to support fine-grained futures in a Java Virtual Machine. Our system is built upon the general thread scheduling system in the JVM and is incorporated with the sampling system which was previously used for dynamic compilation solely. This enables our system to exploit both system resource availability and futures' computation granularity while making inline decisions. While in the previous system, splitting is triggered only by a blocked task or an idle processor. The task granularity is not monitored and considered.

Another effort to support fine-grained futures is called *leapfrogging* [23]. Leapfrogging is a workcrew-style implementation. A task object is created for a future invocation and is put into a task pool. A worker takes a task from the pool and works on them one by one. When a worker is blocked due to some unfinished future, it steals a task that the current task is dependent on and starts to execute the stolen task on top of the current stack. Leapfrogging can be expressed in C's stack frame management mechanism, and thus, it is easier to implement and more portable comparing to LTC. Comparing to our approach, however, it does not consider the granularity of futures, and it has the queue management overhead introduced by its workcrew-style implementation.

There are several previous works related to our synchronization optimization. For example, in [6], there are two clones of each procedure: a fast clone used while the procedure is invoked locally and a slow clone that is used while the procedure is stolen by another processor. In the fast clone, all *sync* operations are translated to *noop* to avoid unnecessary synchronization. Our system is slightly different in that we do not keep two clones of a method. Instead, we use the *onStack* flag which is set dynamically by the future splitter to eliminate unnecessary synchronization. In [5], static analysis is used to eliminate redundant touch operations for futures, which is complementary to our dynamic approach.

Profiling has been used to choose the best parameters of parallel optimizations [4] or the optimal number of threads to use given available system resources [12], etc. In most of these systems, it is assumed that one computation will be invoked repeatedly and the execution will last for a long time. Therefore the system can use several initial runs for learning before making a decision. Our system, however, targets at fine-grained futures, most of which have very short execution time, and usually are not invoked repeatedly. Thus, we use sampling to monitor how long a future has been executed, and to make splitting decision for the current future, instead of its later invocation. We plan to investigate the possibility of exploiting profiling for repeatedly invoked computation as part of future work.

Safe futures proposed in [24] enforce the semantic transparency of futures automatically using object versioning and task revocation so that programmers are freed from reasoning about the side-effects of future executions to ensure correctness of programs. This is complementary to our system and we plan to investigate the performance impact of lazy futures in combination with safe futures as part of future work.

The concept of futures is also employed in distributed environments to optimize task scheduling [8]. Data futures are created to refer to data products that have not yet been created. Their system is similar to our system in the sense of dynamic future scheduling based on cost/benefit estimation. But it is at a much more coarse-grained level with different cost/benefit tradeoffs.

6 Conclusions

As multi-core systems become ubiquitous, we increasingly require programming language constructs that ease parallel programming for developers. *Futures* [17] are one such construct that programmers can use to identify potentially asynchronous computation that can be executed in parallel. Recently, futures have been made available in modern, high-level languages such as Java [11]. However, to ensure

wide-spread use of futures, their implementation must enable efficient and scalable parallel program execution. In addition, avoiding the need for user participation in the decision about when to spawn futures in parallel, and when to execute them sequentially, is critical.

In this paper, we investigate an efficient implementation of futures for Java that automatically, and adaptively decides when and how to spawn futures. Our implementation is the first JVM runtime implementation of futures that couples estimates of future computational granularity (gathered by a low overhead JVM program sampling infrastructure) with underlying resource availability. We refer to our implementation as lazy futures. We empirically evaluate lazy futures using a wide-range of benchmarks, triggers for automatic spawning of futures, processor counts, and JVM configurations. We show that we are able to implement futures for Java in a way that requires no programmer intervention into the spawning decisions of futures and that is scalable and efficient (i.e., comparable to hand-tuned alternatives) for use with fine-grained futures in Java programs.

References

[1] 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.

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

[3] 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 *Proceedings of the 20th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, 2005.

[4] P. C. Diniz and M. C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 71–84, 1997.

[5] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, 1995.

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

[7] 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.

[8] H. A. James and K. A. Hawick. Data futures in discworld. In *HPCN Europe 2000: Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 41–50, London, UK, 2000. Springer-Verlag.

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

[10] BEA JRockit, Java for the Enterprise. http://www.bea.com/content/news_events/white_papers/BEA_JRockit_EntJava_business_wp.pdf.

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

[12] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 236–246, 2005.

[13] D. A. Kranz, J. R. H. Halstead, and E. Mohr. Mul-t: a high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, 1989.

[14] 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.

[15] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001.

[16] J. Plevyak, V. Karamcheti, X. Zhang, and A. A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, 1995.

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

[18] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, June 2001.

[19] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM J. Res. Dev.*, 48(5/6):767–795, 2004.

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

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

[22] 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.

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

[24] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA '05: Proceedings of the twentieth ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 439–453, 2005.