A Case for Cooperative Scheduling in X10's Managed Runtime

Shams Imam
Rice University
shams@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Design, Languages, Performance

Keywords Task Parallelism, Cooperative Scheduling, Delimited Continuations, Async-Finish Model, X10

1. Overview

In this work, we motivate the use of a cooperative runtime to address the problem of scheduling parallel tasks with general synchronization patterns. Current implementations for task-parallel programming models provide efficient support for fork-join parallelism, but are unable to efficiently support more general synchronization patterns that are important for a wide range of applications. In the presence of patterns such as futures, barriers, and phasers, current task-parallel implementations revert to thread-blocking scheduling of tasks. Barriers and futures are two common synchronization patterns advocated by many industry multicore programming models that go beyond the fork-join model, but there is as yet no demonstration of an effective solution to schedule programs with futures and barriers in a scalable fashion when the number of blocked tasks exceeds the number of worker threads.

The use of continuations has previously been proposed for the native (non-managed) C++ runtime for X10 [10]. In that work, continuations are supported by speculatively allocating them on the stack as the program executes. The work-stealing scheduler in their implementation supports the work-first policy prescribed by the Cilk project [1]. Their implementation supports distributed `async-finish` programs along with conditional atomic blocks but does not support X10 clocks (a precursor to Habanero-Java phasers [3]). In our approach, we rely on the help-first policy [6] to create independent stack frames for tasks to enable use of one-shot delimited continuations, as well as the flexibility of using either a work-sharing or work-stealing scheduler. Our cooperative runtime for Habanero-Java (HJ) [8] is general enough to support a wider variety of synchronization constraints than the runtime described in [10].

Thus far, our implementation experience with the cooperative runtime has been with HJ, an explicitly parallel language derived from version 0.7 of X10. HJ has a wide range of synchronization constructs including finish, future gets, data-driven tasks, barriers, and phasers. In our implementation we show how the one-shot delimited continuations and event-driven control primitives can be used to support this wide range of synchronization constructs in a cooperative runtime. These constraints include cases where a single task may trigger the enablement of multiple suspended tasks (as in futures, barriers and phasers). In contrast, current task-parallel runtimes and schedulers for the fork-join model focus on the cases where only one continuation is enabled by an event (typically, the termination of the last child/descendant task in a join scope). Our experimental results show that the HJ cooperative runtime delivers significant improvements in performance and memory utilization on a range of benchmarks using *future* and *phaser* constructs, relative to a thread-blocking runtime system while using the same underlying work-stealing task scheduler.

2. The Cooperative Runtime

In our cooperative runtime [8], when a potential synchronization point is discovered dynamically, thread blocking operations are avoided by suspending the currently executing task and cooperatively scheduling other ready tasks from the work queue. When the suspending condition is resolved, the suspended task (and its continuation) is put back into the work queue to eventually be resumed by a worker thread. Task suspensions are implemented by using standard one-shot delimited continuations (DeConts). This guarantees that the runtime never spawns more worker threads than it was initially started with. The trade-off is that the compiler and the runtime now need to support the overhead of creating the DeConts and handling the tracking of the suspending conditions in addition to the management of threads and tasks.

The runtime cooperatively schedules tasks using DeConts and data structures to resolve suspension conditions in the presence of arbitrary dependences or synchronization constraints. The runtime places tasks into queues while the pool of worker threads continuously attempt to execute tasks dequeued from these queues. Execution of tasks may result in more tasks being spawned and enqueued into the queues. In addition, a list of suspended tasks is maintained with each suspension condition used to implement higher-level synchronization constructs. Resolving suspending conditions moves associated suspended tasks from this list into the ready queue. An application starts with a single *main* task in the work queue which promptly gets executed by one of the worker threads. The application terminates when: *a*) the work queues are empty; and *b*) all synchronization constraints in the program have been satisfied (i.e. no deadlocks).

The work-first policy can be more efficient than help-first for recursive divide-and-conquer parallelism when steals are infrequent [6], the work-first policy cannot be used to support general synchronization constraints. Instead, our runtime uses the *help-first* policy for task scheduling. Under this policy, spawning a child task enqueues it in the task queue and allows the parent task to continue execution past the spawn operation. The child task hence has a stack of its own and can be executed by any of the worker threads. The independent stack allows us to treat the task as a subcomputation and to have a well-defined outer boundary while forming the DeCont. In contrast, using a *work-first* policy [6] does not provide an independent call-stack for a spawned task and requires maintaining fragmented call-stacks to allow helper threads to resume computations. This precludes the use of DeConts in a work-first policy.

With the help-first policy in effect, we wrap the stack of each task around a DeCont which defines an `execute()` method as the continuation boundary. When a worker thread executes a task, it resumes the computation of the DeCont which in turn invokes the `execute()` method (Figure 1). At synchronization points where a task is not allowed to make progress semantically, a DeCont is captured and only the state until the `execute()` method needs to be saved. On returning from a call to `execute()`, the runtime verifies the cause for the return and performs book-keeping if the task was suspended. The worker then goes ahead and tries to dequeue other scheduled tasks to execute and continue making progress towards the overall computation.

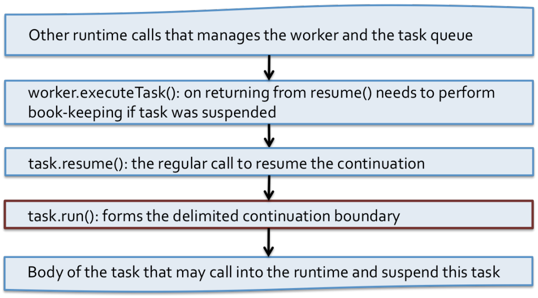
Other runtime calls that manages the worker and the task queue
worker.executeTask(): on returning from resume() needs to perform book-keeping if task was suspended
task.resume(): the regular call to resume the continuation
task.run(): forms the delimited continuation boundary
Body of the task that may call into the runtime and suspend this task

Figure 1: Representation of the runtime call stack when a task is being executed by a worker thread. The `worker.executeTask()` method is responsible for managing the DeConts that may be suspended while executing the body of a task.

On returning to the `worker.executeTask()`, the runtime checks whether the task was suspended and registers the task with the suspending condition. There is no limit to the number of tasks that can be registered to a specific suspending condition. When the suspending condition is resolved, the suspended tasks are rescheduled for execution by placing them back in the work queue. Note that this approach does not use polling to keep track of when the suspended tasks can be resumed. After being scheduled, the queued task is picked up by a worker thread and execution is resumed from the previous suspension point. When the execution of the task completes normally (without suspending) the runtime performs all the cleanup operations associated with the task and then starts looking for more work from the queue.

3. Implementation

We have implemented our cooperative runtime in the HJ language supporting all its available constructs without any changes to the syntax of HJ programs. In HJ, language constructs are supported by a compiler and a runtime system. The compiler generates standard Java class files that run on any standard JVM. Figure 2 highlights the different stages in the HJ compiler. The frontend parses the HJ program and constructs AST nodes to represent the different HJ constructs. The frontend also performs semantic checks for the constructs. The backend then gradually applies a series of transformations, all HJ constructs are later expanded into standard Java operations with calls to the HJ runtime library inserted as needed. For instance, an async is transformed into an instance of an anonymous inner class (closure) that can be passed to the HJ task scheduler. Further details on the general compiler architecture is available at [2]. The runtime is responsible for managing the creation, execution, and termination of tasks using a variety of task scheduling policies. The decision of which policy to use must be made at compile time since the generated bytecode is tailored to the appropriate scheduler.

Our implementation for the cooperative scheduling policy adds two additional stages in the backend. This is to translate calls to *pausable* methods (Section 3.2) and perform bytecode transformation (Section 3.1), as shown in Figure 2. The cooperative runtime system is responsible for scheduling the tasks on a (user configurable) fixed number of worker threads. In our implementation, we extend the compiler to generate code that detects suspension points and generates code for the tasks. We rely on the help-first policy to have independent stack frames for tasks. As a result, child tasks do not share the call stack with their parents and this enables us to easily use DeConts. These continuations are resumed exactly once which allows us to have an efficient implementation compared to general continuations.

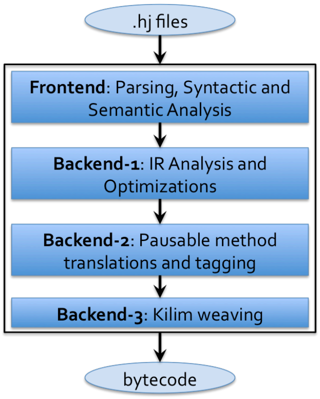
.hj files
Frontend: Parsing, Syntactic and Semantic Analysis
Backend-1: IR Analysis and Optimizations
Backend-2: Pausable method translations and tagging
Backend-3: Kilim weaving
bytecode

Figure 2: Architecture of the HJ cooperative compiler.

Our implementation conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for continuations or for storing away and restoring the stack. In [4], Drago et al. extend a custom VM to support continuations. However, the delimited and one-shot continuations they support fail if a thread calls a continuation captured by another thread. Instead we choose a solution where DeConts are thread independent and can be restored on any thread.

3.1 Delimited Continuations - Kilim Weaver

The alternative to support DeConts on a standard JVM is to transform the bytecode in class files. We use a slightly modified version of the open source bytecode weaver provided by the Kilim framework [9] to support DeConts. Our extension to Kilim allows for DeConts to be captured and resumed on custom threads (e.g. provided by the HJ runtime) rather than on threads managed by the Kilim runtime.

The Kilim weaver works by transforming code of methods which can possibly suspend. It recognizes such methods by the presence of a `Pausable` exception in the method signature. It is important to note that no actual exceptions are thrown or caught to minimize the overhead of capturing and resuming continuations. Instead, the transformation performed is similar to a continuation passing style transform, except that only methods that can possibly suspend are transformed. In HJ, users do not annotate their methods with such an exception. The HJ compiler is responsible for tagging possible suspendable methods. We explain the tagging mechanism of our implementation in Section 3.2.

While transforming suspendable methods, each method gets an extra argument of type *fiber*. The fiber is a data structure used to save the state of the stack including local variables while capturing a continuation. While resuming the continuation, the call stack is restored using the control flow information saved in the fiber. The weaver does handle the capturing continuations running inside `synchronized` blocks. This does not concern us because synchronized blocks are neither used in our runtime nor are they allowed in HJ programs. However, we do use the weaver's support for the handling of `try-finally` blocks. The weaver also includes other optimizations such as lazily restoring the continuation frames in the call chain, custom state objects to store local variables, and avoiding store of variables which will no longer be referenced into state objects.

3.2 Tagging Suspendable Methods

The Kilim weaver relies on the presence of the Pausable exception in the list of thrown exceptions of a method declaration to identify which methods to transform. In the HJ cooperative runtime, the continuations are created transparently to the user and the user is not required to annotate method signatures with exceptions in her code. Hence, the HJ compiler needs to insert the Pausable exception into the declarations of user written methods which may suspend. The HJ compiler first translates the user written code, which uses various synchronization constraints, into calls to suspendable methods provided by the cooperative runtime. After this transformation, the code in Java class files is no longer valid as there are method bodies in user-written code which throw the checked Pausable exception without declaring it in the throws clause of the method.

Next, the compiler detects calls to suspendable methods in the user code and incrementally annotates all the user methods with the Pausable exception. The algorithm for tagging suspendable methods basically does a reachability analysis for the Pausable exception and is as follows:

- Overridden methods of classes provided by the runtime or standard HJ/Java API do not change their suspendable behavior. They preserve their defined signature to avoid the weaver having to possibly transform a large number of API classes.
- Methods are suspendable if they invoke other suspendable methods. This can recursively cause the signatures in user written parent classes to change and become suspendable.
- Similarly, interface methods in user written classes are suspendable if any of the implementing classes have method bodies invoking other suspendable methods.
- Constructors and undetermined static methods are not suspendable.
- All remaining methods are conservatively tagged as suspendable.

This algorithm terminates when it has successfully tagged (either as suspendable or as not suspendable) all user-written methods or if an error occurs. If during the tagging process there is a conflict, such as the implementation of an interface method from the HJ API in the user-written code contains a call to a suspendable method, the HJ compiler fails and returns the appropriate error message. Once the annotation completes without error, the weaver can be run to perform the transformation required to support continuations. After successful compilation, the generated bytecode can be run on any standard JVM by including the HJ runtime in the classpath.

4. Preliminary Results

We compare the performance of our cooperative runtime in HJ with the work-sharing runtime of HJ and the managed runtime of X10. We include barrier benchmarks from the IMSuite benchmark suite [7] which include implementation of various benchmarks implemented both in HJ and X10. To compare the performance of futures we use the Smith-Waterman benchmark [11] and the Binary Trees benchmark, from the Computer Language Benchmarks Game [5].

4.1 Experimental Setup

The benchmarks were run on a 12-core 2.8 GHz Intel Westmere SMP node with 48 GB of RAM per node, running Red Hat Linux (RHEL 6.0). Each core has a 32 KB L1 cache and a 256 KB L2 cache. The software stack includes Java Hotspot JDK 1.7, Habanero-Java 1.3.1, and X10 2.3.1-2. Each benchmark used the same JVM configuration flags (-Xmx8192m). It was configured to use 12 worker threads (export X10_NTHREADS=12 for X10 and -places 1:12 for HJ), and was run for five iterations. The bar charts show the arithmetic mean of the five runs, and the error bars represent one standard deviation. All benchmarks use the same single-place shared memory algorithm in their implementation. The parallelism related constructs of HJ are similar to that of X10 with minor differences in syntax and semantics. For example, HJ constructs isolated, next, and phaser map to the atomic, Clock.advanceAll and clock constructs in X10.

Figure 3 shows execution times on a log scale for the *future* and the *barrier* benchmarks with the HJ Cooperative runtime, HJ Work-sharing runtime, and X10 Managed runtime. In the Binary Trees benchmark, there is a relatively larger delay between the creation of the future and the attempt to resolve its value. This nature of the benchmark allows the blocking schedulers in HJ and X10 to make some progress in executing the futures and thus helps in minimizing the blocking operations due to calls on unresolved futures. Even with this property, the cooperative runtime outperforms the blocking version of the HJ work-sharing runtime, and the X10 managed version is over an order of magnitude slower.

In the Smith-Waterman benchmark, futures are used to represent the value at each cell of the dynamic programming *table* and backtracking starts at the highest corner cell. Each cell depends on values from three neighboring cells and thus each cell has exactly three suspension points, once for each attempt to resolve the future of a neighboring cell, while computing its own value. Due to the comparative lack of delay while trying to resolve a future after its creation, many blocking operations are performed in the blocking runtime. This degrades performance and the cooperative version outperforms the blocking version by a factor of 3× even on small inputs. The X10 version fails at runtime with the “Place(0): TOO MANY THREADS” error.

The next set of benchmarks are graph algorithms from the IMSuite benchmark suite [7]. We converted the distributed X10 implementation to run on a single place by removing the DistArrays and at clauses. The IMSuite characterization covers aspects of task parallel programs such as number of synchronizations (using clocks or phasers), asynchronous task creation, task termination and atomic operations. Since we used an input graph size of 512, most of the kernels create 512 asynchronous tasks that frequently participate in barrier operations for synchronization. The runtime has to compensate for the blocked threads by creating additional worker threads. Since there are only 12 cores available, the creation of additional threads in the blocking versions (HJ Work-sharing and Managed X10) cause performance loss compared to the HJ Cooperative version which runs the applications on the 12 worker threads.

BFS-BF and BFS-Dijk also had X10 implementations using SPMDBarrier which performs better in shared-memory synchronization contexts. BFS-BF completed in 76.39 ms and BFS-Dijk in 240.79 ms respectively (i.e. better than the HJ cooperative runtime), but required us to set the number of worker threads to 512 (export X10_NTHREADS=512). We also had X10 implementations using SPMDBarrier for DomSet, MIS and MST but they failed to complete successfully appearing to be in a deadlock. The issue with using SPMDBarrier is that the number of worker threads need to be configured to the number of tasks that will participate in the barrier. This approach is unscalable as larger input program would require setting larger values for the number of worker threads. For example, using a 1024 node input graph (i.e. requiring export X10_NTHREADS=1024) throws an OutOfMemoryError as the X10 runtime is unable to create 1024 native threads.

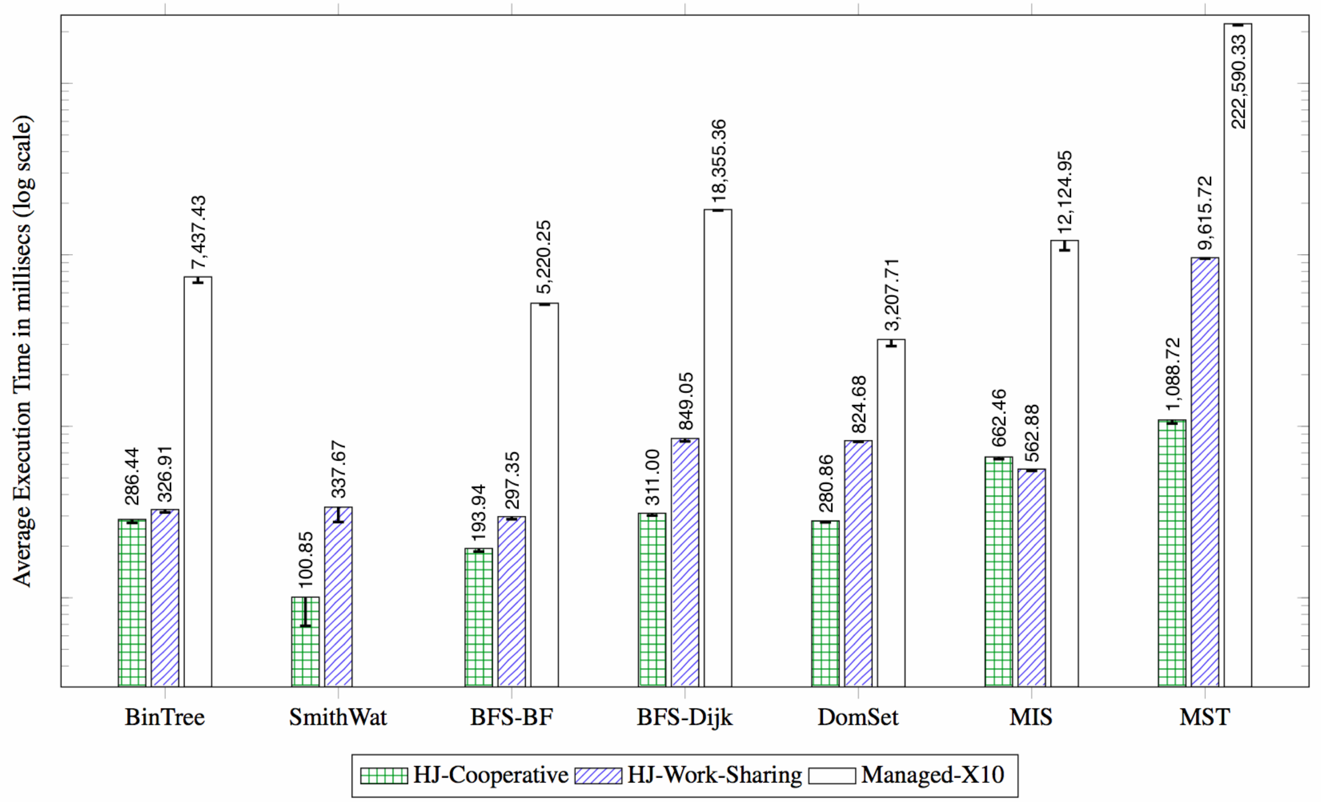

Figure 3: Results for future and barrier benchmarks with HJ Cooperative runtime, HJ Work-sharing runtime, and X10 Managed runtime. The Binary Tree (BinTree) benchmark operating on a tree with depth of 10. Smith Waterman (SmithWat) on strings of length 119 and 107. BFS-BellmanFord (BFS-BF), BFS-Dijkstra (BFS-Dijk), Dominating Set (DomSet), Maximal Independent Set (MIS) and Minimum Spanning Tree (MST) benchmarks from IMSuite with an input graph of size 512 nodes and artificial load values set to 0. Execution times are reported in log scale, the actual execution time in milliseconds of the benchmark is reported next to the bars.

References

[1] 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 *Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. .

[2] V. Cavé, J. Zhao, Y. Guo, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ*, pages 51–61, 2011.

[3] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[4] I. Drago, A. Cunei, and J. Vitek. Continuations in the Java Virtual Machine. In *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2007.

[5] B. Fulgham. binary-trees benchmark. URL http://benchmarksgame.alioth.debian.org/u32/performance.php?test=binarytrees.

[6] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. .

[7] S. Gupta and V. K. Nandivada. IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. *CoRR*, abs/1310.2814, 2013.

[8] S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *Proceedings of the 28th European conference on Object-Oriented Programming* , ECOOP '14, New York, NY, USA, 2014. ACM.

[9] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. .

[10] O. Tardieu, H. Wang, and H. Lin. A Work-Stealing Scheduler for X10s Task Parallelism with Suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. .

[11] Wikipedia. SmithWaterman algorithm. URL http://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm.