Leapfrogging: A Portable Technique for Implementing Efficient Futures

David B. Wagner Bradley G. Calder

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Abstract

A *future* is a language construct that allows programmers to expose parallelism in applicative languages such as MultiLisp [5] with minimal effort. In this paper we describe a technique for implementing futures, which we call *leapfrogging*, that reduces blocking due to load imbalance. The utility of leapfrogging is enhanced by the fact that it is completely platform-independent, is free from deadlock, and places a bound on stack sizes that is at most a small constant times the maximum stack size encountered during a sequential execution of the same computation. We demonstrate the performance of leapfrogging using a prototype implementation written in C++.

1 Introduction

A *future* is a language construct that enables programmers to identify parallelizable computations in an otherwise sequential program. Rather than waiting for the result of such a computation, the program receives a "placeholder" for that result and is able to continue executing. The placeholder behaves just like any other variable until an attempt is made to use its value; at that point, if the computation is not finished then the thread of execution trying to obtain the value will be blocked until the value is ready.

The principal design 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 [of work] will take place to the runtime system" [6].

Another nice feature of futures is that all synchronization is implicit. The programmer never explicitly checks to see if the value of a computation is ready; if it is not, the thread of control trying to use that value is transparently blocked. Because of this, futures are much easier to use than e.g., fork-join.

There are significant challenges to making a futures runtime system efficient when the emphasis is on *minimal programmer effort*. For example, the programmer might identify very fine-grained computations as candidates for parallel execution. Actually executing each of these computations as a separate task might be counterproductive, since the overhead of creating tasks and synchronizing might overwhelm the time saved by parallelization.

Another potential problem is that of unbalanced computation graphs. The computation graph might be structured such that the result of a future is almost never ready when called for; this

leads to extra overhead, either in the form of idle processor time or context switching. Even worse, a careless implementation can lead to very subtle deadlock problems in realistic (not contrived) computations.

A very elegant, but complicated, solution to these problems was presented by Mohr, Krantz and Halstead [6]. Their system, which we will explore in more detail in the next section, is based on *continuations*.

In this paper we present an alternative technique for implementing efficient futures, *leapfrogging*, that also is very portable. Our testbed system is currently implemented in `C++`, although the leapfrogging technique is completely language- and platform-independent. The crucial point is that leapfrogging provides efficiency similar to that of more complex techniques such as LTC, but can be ported with very little effort to any system that provides process creation, shared memory, and lock synchronization primitives.

2 Performance Issues

2.1 Basic issues

The `sum-tree` example from [6][1] provides a good illustration of how difficult it is to get good load distribution and minimal blocking using futures, because it contains a plethora of potential parallelism. A naive approach to parallelizing `sum-tree` would use a separate task for each future, so that the number of tasks used would be equal to the number of leaves in the tree. This excess task creation will greatly diminish the speedup of the parallelized program, since the granularity of task creation will certainly be much larger than the granularity of a leaf computation.

In order to avoid excess task creation, the programmer could limit the creation of tasks to a fixed depth in the algorithm itself. Below that depth the algorithm would execute the recursive calls directly, rather than futurizing them. In this way, the programmer takes responsibility for matching the granularity of the tasks to the overheads of the runtime system. Unfortunately, it is frequently the case that the only way to find the optimal cut-off depth is through experimentation.

One possible attack on the granularity problem is called *load-based inlining*. The idea is to dynamically monitor the load and create a new task to compute a future only when processors are idle. Otherwise, the future is executed inline by the task that creates it. This keeps the number of tasks created close to the number of processors available. Qlisp [3, 4] provides the programmer with primitives that inspect the state of the system, and allows the programmer to specify an arbitrary predicate to control the inlining of futures.

One problem with any type of inlining is that a decision to inline a future cannot be revoked. This could result in some processors sitting idle while others work on large tasks, especially if the task granularity is unpredictable. Another problem is that keeping track of processor availability creates extra overhead for the task scheduler.

Thus, there appears to be a fundamental tradeoff between achieving good task granularity and achieving good load balance.

[1] `Sum-tree` is a routine that recursively sums the values stored in the leaves of a binary tree.

2.2 WorkCrews

The "obvious" solution to the granularity vs. load balancing dilemma is to note that there is really no need to create a separate task for every future. Instead, a WorkCrew-style approach can be adopted [7]: a future is implemented as a passive object that contains enough information to carry out the computation. These passive objects are then picked up by worker tasks and executed. The number of workers is fixed and is equal to the number of processors. The advantage of using passive futures and active workers is that the overhead of creating a future is reduced so much that inlining, with its associated load balancing problems, rarely is required for good performance.

Unfortunately, there are other performance problems with a WorkCrew-style implementation. Foremost among these is that of workers blocking to wait for an unfinished future. In the one-task-per-future implementation, either (a) the blocked task can idle its processor until the desired result is available, or (b) it can context switch to some other task. The former lowers processor utilization, thus the latter alternative would be preferred unless futures were of extremely small granularity. Unfortunately, in the WorkCrew approach there is no other task to switch to!

We conclude that a naive WorkCrew approach would be considerably faster than the one-task-per-future approach for balanced computations, but could be arbitrarily bad for unbalanced computations.

2.3 Lazy Task Creation

Mohr et al. [6] have devised a clever scheme that they call *Lazy Task Creation* (LTC) that solves the problems just discussed. Their technique is built into a compiler for Mul-T, a parallel dialect of Scheme.

In Mul-T, `(K (future X))` is interpreted to mean, "Start evaluating X in the current task, but save enough information so that its continuation K can be moved to a separate task if another processor becomes idle" [6]. This effectively makes the decision to inline any future a revocable one, so the default behavior is that every future is inlined. There are exactly as many tasks as processors, and continuations are *stolen* (i.e., moved to another processor) only when a processor becomes idle.

LTC is implemented by maintaining a list of continuations on the producer task's stack. Then when a continuation is stolen, the scheduler cuts the stack apart and replaces the inlined future's stack frame with a placeholder. The continuation-based strategy also provides a strategy for dealing with the problem of a task blocking on some unfinished future. When such a case arises, the scheduler selects another continuation from the blocked task's stack to work on and cuts the stack again. The authors claim that the costs of doing this are low enough to meet their performance goals. For more details see [6].

Obviously, LTC is a completely general approach with the potential for very good performance. If the load is well-balanced, continuation-stealing will happen infrequently, if at all, and the computation will proceed much faster than if the futures had not been inlined. On the other hand, if the computation is seriously unbalanced, then it is crucial that the implementation make the time required to steal a continuation competitive with having created a separate future in the first place.

The principal drawback to the LTC alternative, as we see it, is that it requires some very intricate implementation. Using a traditional call-stack implementation, continuation stealing requires the copying of stack frames, which is platform-dependent as well as expensive. To solve the latter

problem, Mohr et al. implemented a linked-list call-stack, which exacerbates the former problem. Practically speaking, this will likely limit the use of LTC to a narrow community of users.

Our experience in [8] has convinced us that futures have a much wider domain of application than they have usually been credited with, and our motivation is thus to "spread the word" about them to as wide a community of users as possible. The main contribution of this research is a new technique, *leapfrogging*, that provides very good performance while being extremely easy to implement.

2.4 Leapfrogging

The leapfrogging technique is built into a WorkCrew-style implementation. In this implementation, futures are passive objects rather than active tasks. The number of worker tasks is fixed and equal to the number of processors, and each worker has a corresponding FIFO work queue into which futures are inserted and removed. A worker can remove a future from any work queue, but by default inserts futures only into its own.

When a worker finishes evaluating a future and has nothing in its work queue, it looks in other workers' work queues in order to find work. This is termed *stealing*. Since futures typically are not inlined (because the cost of creating them is so small), if there is work available anywhere in the computation, an idle worker eventually will find it.

Stealing futures is the principal way of balancing the load among the workers, but it does not solve the problem of workers blocking on unresolved futures. *Leapfrogging* is our solution to the problem of worker blocking. A prototypical leapfrogging scenario is depicted in Figure 1.

Suppose that a worker, X, creates a future F1 and begins executing its continuation K1 (Figure 1(a)). Now worker Y steals F1 and the scenario becomes that of Figure 1(b). (In the figure, the clear nodes represent uncomputed futures, and the opaque nodes represent work that has been or is currently being executed. The current position of each worker is marked with an arrowhead, and the flow of control of each worker is differentiated by line styles.)

Sometime later, the computation might appear as shown in Figure 1(c). Both workers have created additional futures and are further down in the computation tree. At this point, suppose that X returns to the next higher level in the tree and requires the value of F2, which has not yet been computed. Since no other worker has claimed F2, X would inline it (Figure 1(d)).

Eventually, X will return to the root of the tree and require the value of F1. If the load is unbalanced, it may be the case that F1 is not ready. Rather than block itself, X takes the first future in Y's work queue that is descended from F1 (F3) and starts executing it. The result is shown in Figure 1(e).

The motivation for leapfrogging is that X cannot proceed until Y has completed evaluating F1, so X is indirectly helping itself if it evaluates F3. The assumption implicit in this statement is that F1 depends on the value of F3; in other words, that a future does not create other futures unless it is going to use their values. This is certainly true for recursive, AND-parallel computations. For other types of computations, this assumption may be violated.[2]

Of course, it is perfectly plausible that after X leapfrogs Y but before X finishes F3, Y might require the value of F3. In this case, if X has created yet another future (F4) then Y can leapfrog over X, leading to the situation depicted in Figure 1(f). This can continue for the entire depth of

[2]For example, this assumption will not hold for OR-parallel computations, e.g., certain search algorithms. We do not consider OR-parallelism in this paper.

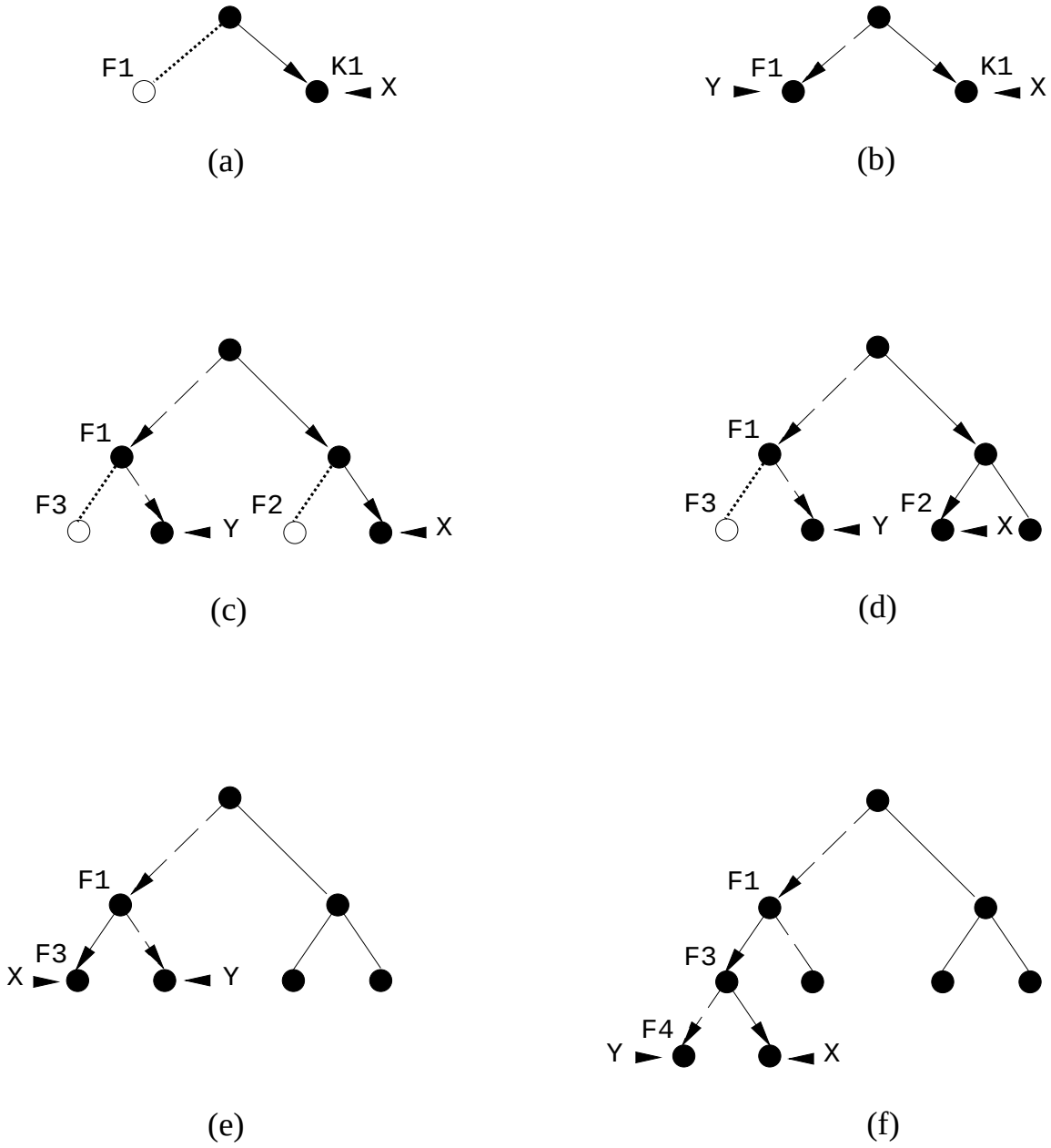

Figure 1: Typical leapfrogging scenario.

the tree, if necessary; by now it should be apparent why we chose the term "leapfrog" to describe this mechanism.

The action of leapfrogging prevents workers from idling when there is work to do. The difference between stealing and leapfrogging is that the former is done by an idle worker (with an empty stack), whereas the latter is done by a worker that is trying to resolve a future (whose stack is in some intermediate stage of a computation). We note, however, that all of the futures on a worker's stack always lie on a single path from the root of the computation tree to a leaf, although there may be any number of "gaps" along the way. This implies that (a) the computation cannot deadlock unless there is a cycle in the dependence graph, and (b) no worker's stack will ever exceed the maximum stack size that a sequential execution of the program would cause.[3]

It might seem to be more beneficial in the long run to allow worker X to leapfrog onto a future as high up in the computation tree as possible, because it would likely have a larger granularity. However, this would violate the conditions necessary to ensure the bounded stack property.

2.5 Comparison with Lazy Task Creation

The creating of a future in our system is similar to Mohr et al. idea of creating a continuation because our futures are just passive objects on a work queue, and if no other processor steals the futures the worker that created them executes them. The main differences are that in our system, a newly created future is saved for later execution and the worker that created the future continues executing the current function (the continuation). In Mul-T's lazy task creation approach the future is executed immediately and the continuation is saved on the task's stack.

Although the costs of creating a future in our system and of creating a continuation in Mul-T appear to be comparable, the cost of *stealing* a continuation is much higher than the cost of stealing a future because of the stack manipulations that are necessary to accomplish the former. On the other hand, if the computation tree is balanced then stealing (of continuations or futures) will be a relatively rare event.

Mul-T may have an advantage over our system in the case of a balanced computation tree. Consider Figures 1(a) and 1(b) once again, and suppose that the amounts of computation in F1 and K1 are approximately the same. In our system, since worker X will get a "head start" on worker Y, it is quite likely that X will finish K1 before Y finishes F1, causing X to leapfrog over Y, or to block if there is no available future in Y's work queue.

In Mul-T, the roles of X and Y in the diagrams would be reversed, with X executing F1 and Y executing K1. Assuming that X again finishes before Y, X does not need to wait for Y, since Y has stolen the continuation. In other words, it will be Y, and not X, that must eventually return to the next higher level in the tree. We note, however, that the cost of leapfrogging is small (only slightly larger than the cost of stealing a future), so in the case that there is an available future in Y's work queue, our system is really not at a disadvantage relative to Mul-T.

On the other hand, if the computation tree were unbalanced it might transpire that Y would finish before X. In our system, this would free Y to look for work elsewhere. But the Mul-T strategy in this case is to have Y steal a continuation *from itself*. However, we cannot envision a natural scenario in which there would be any continuations in Y's continuation list under such circumstances. It is not clear, then, whether or not Mul-T would have any choice but to block Y.

[3] Refer to Section 3 for a more precise discussion of these properties.

In some sense, the two scenarios described above are the dual of each other. All in all, the two approaches have comparable functionality and (probably) comparable performance. But we maintain that our approach is much easier to implement and, as a corollary, is more portable.

3 Implementation Details

3.1 Synchronization

A future can be in one of three states: not executing (NE), executing (E), or finished (F). The meanings of these states are:

NE — The future is not yet evaluated, nor is any worker in the process of doing so.

E — The future is being evaluated by some worker, but the value is not ready.

F — The future has been computed.

There is assumed to be one work queue for each worker, and each work queue is protected by a lock. All state changes take place while one of these locks is being held. When it becomes necessary to wait for a state change in a future, a worker spin-waits on the future's state variable. This is quite efficient on a multiprocessor with coherent caches.

Every future contains a `workerid` and a `creatorid`. `Creatorid` is set to the integer index of the worker in whose work queue the future was placed when it was created. `Workerid` is set to the integer index of the worker that is evaluating the future, if any.

Every future also contains a `depth` field indicating its depth in the computation graph. Furthermore, each worker keeps track of a value `current_depth`, representing that worker's current depth in the graph.[4] The depth of the main routine is by definition 0. Each time a worker creates a new future, it sets the depth of that future to `current_depth` + 1.

When a worker executing a future `f` tries to resolve a future `q`, the action taken depends on the state of `q`:

- If `q.state` is NE, the worker finds the work queue in which `q` resides using the `q.creatorid` field, and locks that work queue. If `q.state` is still NE, the worker dequeues `q`, sets `q.workerid` to its own id, sets `q.depth` to max(`q.depth, current_depth+1`), sets `q.state` to E, and unlocks the work queue. If not, it proceeds to the appropriate case.
- If `q.state` is E, the worker attempts to leapfrog worker `q.workerid`. The worker is allowed to leapfrog if it can find some future `g` in worker `q.workerid`'s work queue with a depth that is *strictly greater* than `max(f.depth, q.depth)`. If so, it attempts to dequeue `g` and evaluate it. It continues this behavior until `q.state` changes to F.
- If `q.state` is F, the worker obtains the value.

The constraint `g.depth` > `max(f.depth, q.depth)` in the leapfrogging case is called the *leapfrog depth rule*. We emphasize this because it is a necessary condition to guarantee freedom from deadlock and bounded stack sizes.

[4] Note that it is necessary for a worker to increment `current_depth` each time a statically inlined call to a futurized function is made. One way to implement this (in C++) is to have the programmer encapsulate the futurized function in a subclass of a special base class, which in turn overloads the *member selection operator* [2]. This enables the base class to track every call to the function.

3.2 Deadlock freedom and stack size bounds

Given two very simple (and reasonable) assumptions about program behavior, we can guarantee the following desirable properties about our implementation:

P1: The computation will be deadlock-free.

P2: The maximum stack size (measured in stack frames) of any worker will be at most a constant factor times the maximum depth of any node in the computation graph.

The reason for the "constant factor" disclaimer in P2 is that, while the maximum number of activations of a futurized function on a worker's stack will never exceed the maximum depth of the computation graph, there may be an extra stack frame or two *per activation* because of calls to runtime system utility functions (function wrappers, type coercion operators, etc.).

The assumptions that allow us to guarantee P1 and P2 are:

A1: The precedence graph for the computation is acyclic.

A2: If `f1` creates `f2` then `f1` depends upon f2.

The necessity of A1 is obvious; however A2 is more subtle. A2 implies that a future does not create "dangling" futures, i.e., futures that are still unevaluated when their creator returns. This assumption is certainly true of all recursive computations, and many non-recursive computations as well.

Space constraints prevent us from including any proofs here; the full proofs can be found in [8]. We show that, under the assumptions given, the leapfrog depth rule not only is a sufficient condition for P1 and P2 to hold, but also is a necessary one.

3.3 Portability

It should be clear from the discussion in this section that our techniques are not platform- or language-dependent. In particular, we do not require knowledge of stack-frame layout.

The parallel processing requirements of the software are satisfied by any system that provides a mechanism for thread creation (Cthreads, Posix threads, even heavyweight processes with shared memory such as are provided by Dynix's `m_fork()`) and locks. Our current implementation is based on Cthreads [1].

The techniques are most efficient using spin-waiting locks, but the implementation can be modified easily to work with blocking locks. All lock routines are accessed indirectly through macros that are declared in a single header file. For example, on the Sequent we changed the macros to allow the program to use native Dynix spinlocks rather than the slower locks provided by our Cthreads runtime system. This took only a few minutes and resulted in a noticeable improvement in performance.

4 Experimental Results

The results presented in this section were obtained from executions of our benchmark programs on a Sequent Symmetry model B. We recorded the timings for each benchmark using 1, 2, 4, 8 and 16

processors. In doing multiple timings for each run we found that the timings were very consistent on all benchmarks.

We graph not only at the speedup of our futurized implementations relative to the sequential implementation (denoted by (S) on the graphs), but also relative to the futurized implementation running with one worker on one processor (denoted by (F) on the graphs).

We have found that recursive programs can benefit from load-based inlining, using a work queue limit of around two to four. A small work queue limit is reasonable because our recursive programs create futures at a steady rate. If future creation were extremely bursty then such a small work queue limit could cause poor performance. The only way to *guarantee* good performance with load-based inlining is through experimenting with some work queue limits until one is found that best fits the application.

4.1 A synthetic benchmark

In order to test out how our futures perform with different granularities we ran a synthetic benchmark, similar to the one in [6].

The synthetic benchmark is a modification of the `sum-tree` program. The only difference is that before a leaf returns its value, a for loop is executed that delays the leaf node for a specified number of machine instructions. The average number of machine instructions per future is actually half of this number, because the loop is executed only in the leaf futures and they account for only half of the futures in the computation tree.

The granularity benchmark was run with 16 processors (and thus 16 workers). Efficiency is the sequential time divided by the product of the number of processors and the parallel time. As can be seen in Figure 2, small granularities leads to poor efficiency because of the overhead associated with creating futures, although load-based inlining yields a substantial improvement. However, efficiency exceeds 90% for granularities larger than about 750 instructions.

Our performance is comparable to but not quite as good as the Encore implementation of Lazy Task Creation reported in [6]. However, this is somewhat misleading, because certain futures-related overheads, which are built into the Mul-T compiler, are present in the sequential times reported in that paper; these overheads reportedly cause dilations of the execution times of sequential programs in the range of 1.4 to 2.2 [6, Table 1].

4.2 Gamma

Gamma is a recursive, adaptive quadrature algorithm that computes the gamma function:

$$\Gamma(n) = \int_0^\infty x^n e^{-x} dx$$

The basic subroutine in Gamma computes the area under the integrand over a given interval using trapezoidal rule. If the curvature of the integrand in the interval is above a certain threshold, the interval is bisected and recursive calls are made. Because the integrand in the gamma function has widely varying curvature, the computation tree is highly unbalanced, leading to a great deal of leapfrogging. Thus, Gamma is a good test of the efficiency of leapfrogging.

Figure 3 shows that we get close to linear speedup, and we do not lose much speedup in comparison to the sequential program.

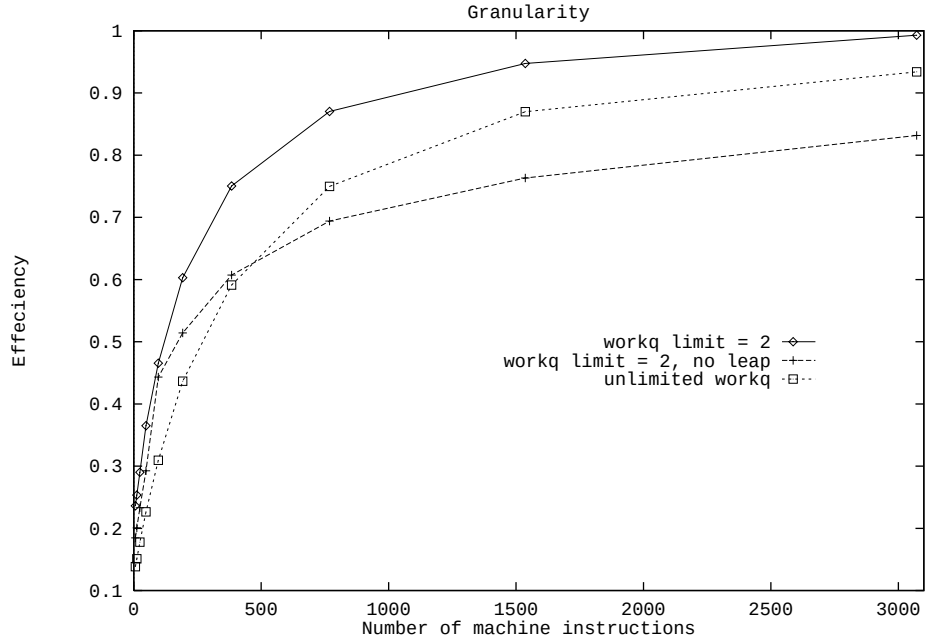

Figure 2: Efficiency vs. granularity on the synthetic benchmark.

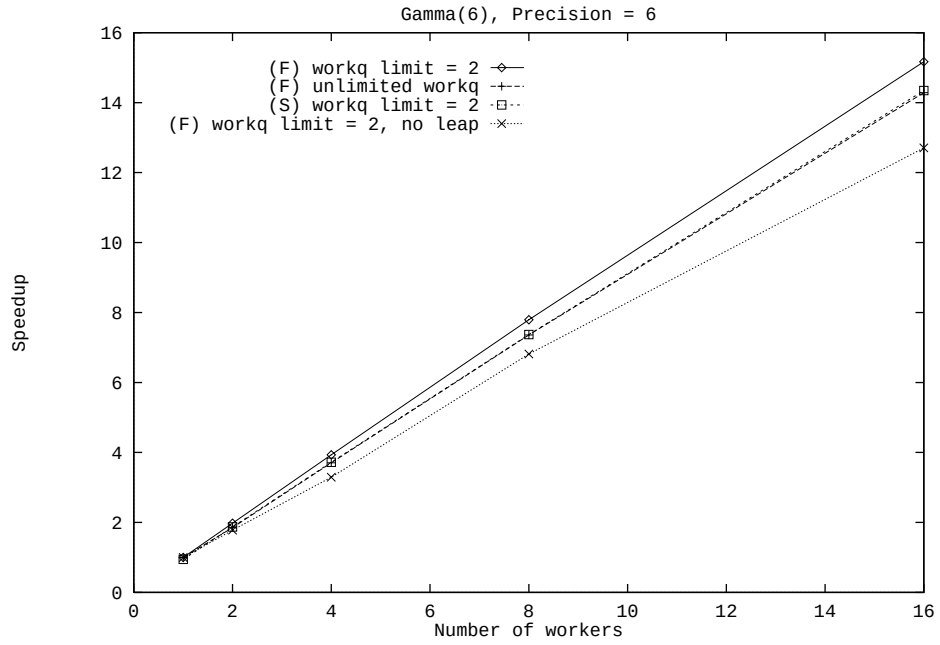

Figure 3: Speedup for Gamma.

In the graph there are two different futurized executions being compared, one that uses load-based inlining and another that has unbounded work queues. For the load-based inlining we fixed the work queue sizes at two, a number we arrived at through trial and error. This is one benchmark in which load-based does not appear to offer much improvement.

4.3 Queens

Our next benchmark, Queens(N), recursively counts all the possible solutions to the problem of placing N queens on an NxN chess board without any queen attacking another.

The Queens problem is interesting because the programmer does not know in advance how many futures a given recursive call might create. If the programmer creates a a future for every recursive call, all of the futures may be stolen by other workers before the current worker gets a chance to evaluate any of them. In such a scenario, the worker will almost certainly either block or leapfrog when it tries to resolve those futures.[5]

A better approach is what we call *save first recursive future* (SFRF). Using SFRF, when the program encounters the first recursive call it saves the parameters and sets a flag rather than creating a future. All subsequent recursive calls for that row are turned into futures. Once the entire row has been evaluated for possible queen placings, the flag is checked and if it is set the recursive call is done.

Figure 4 shows the speedup of the queens program using SFRF with a work queue size limit of four. The futurized code when compared with itself (F) gets almost linear speedup up to eight processors. A possible reason for the smaller slope between eight and sixteen processors is that the granularity of the futures varies from $O(n)$ to $O(n^2)$. Thus, during execution there might be times where there is not enough work to keep all 16 processors busy.

5 Conclusions and Future Directions

The principal contribution of this research is *leapfrogging*, a portable, efficient implementation technique for futures. Leapfrogging handles the problem of worker blocking, balances the load well and is completely platform-independent. In addition, it is provably free from deadlocks and worker stack sizes are kept within a constant factor of the maximum stack size that a sequential execution of the computation would encounter.

We have demonstrated the performance of leapfrogging using a prototype system implemented in C++ using Cthreads [1] to manage parallelism and synchronization. However, the implementation is easily portable to any platform that provides thread creation primitives, shared memory, and mutual-exclusion locks.

Additional information on our implementation and our experiences using futures on a variety of parallel program structures can be found in the unabridged version of this paper [8].

References

[1] Cooper, E., and Draves, R. C-Threads. Tech. Rep. CMU-CS-88-154, Carnegie-Mellon University, Feb. 1988.

[5] This is one scenario in which LTC has a clear advantage over leapfrogging.

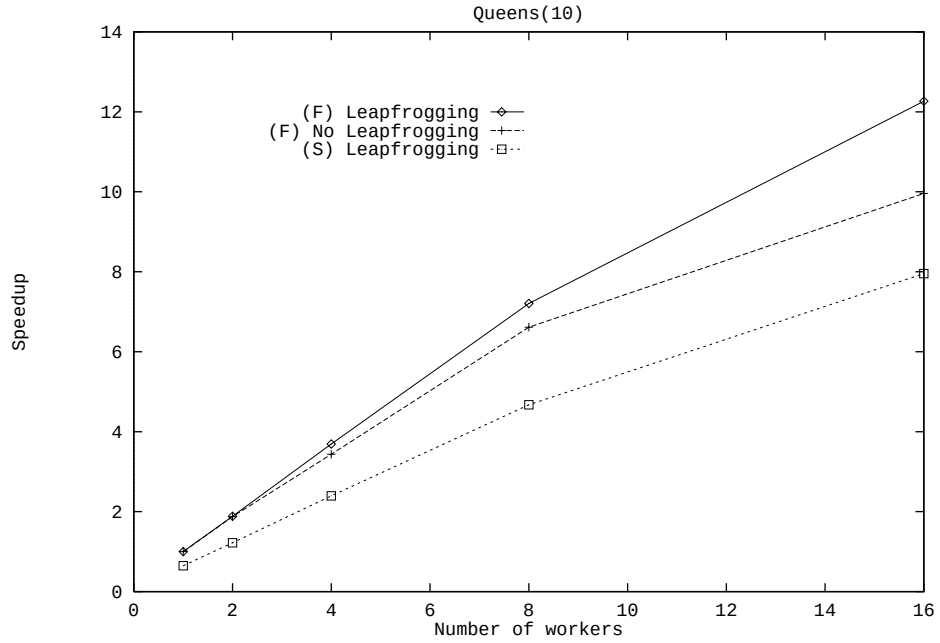

Figure 4: Speedup for Queens(10).

[2] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[3] Goldman, R., and Gabriel, R. Preliminary results with the initial implementation of Qlisp. In *Proc. 1989 Conf. on Lisp and Functional Programming* (July 1989), ACM SIGPLAN, ACM, pp. 143–152.

[4] Goldman, R., and Gabriel, R. Qlisp: Parallel procesing in Lisp. *IEEE Software* (July 1989), 51–59.

[5] Halstead, Jr., R. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 501–538.

[6] Mohr, E., Kranz, D., and Halstead, Jr., R. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991), 264–280.

[7] Vandevoorde, M. T., and Roberts, E. S. Workcrews: An abstraction for controlling parallelism. *Int. Journal of Parallel Programming 17*, 4 (1988), 347–366.

[8] Wagner, D., and Calder, B. Portable, efficient futures. Computer Sci. Dept. Tech. Rep. #CU-CS-609-92, University of Colorado, Boulder, CO, Aug. 1992. (To be submitted, probably to IEEE TPDS).