Work-Stealing by Stealing States from Live Stack Frames of a Running Application

Vivek Kumar[†] Daniel Frampton[†] David Grove[‡] Olivier Tardieu[‡] Stephen M. Blackburn[†]

[†]Australian National University [‡]IBM T.J. Watson Research Center

Abstract

The use of a work stealing scheduler has become a popular approach for providing task parallelism. It is used in many modern parallel programming languages, such as Cilk and X10, which have emerged to address the concerns of parallel programming complexity on modern multicore architectures.

There are various challenges in providing an efficient implementation of work-stealing, but in any implementation it must be possible for the thief to access the execution state required to perform the stolen task. The natural way to achieve this is to save the necessary state whenever a producer creates stealable work. While the ability to provide some degree of parallelism may dominate performance at scale, it is common for the vast majority of *potentially* stealable work to never actually be stolen, but instead processed by the producer itself. This indicates that to further improve performance we should minimize the overheads incurred in making work available for stealing.

We are not the only ones to make this observation, for example X10's current C++ work-stealing implementation stack-allocates state objects and lazily copies them to the heap to avoid unnecessary heap allocation during normal execution. In our context of a Java virtual machine, it is possible to extend this idea further and avoid heap allocating state objects, but instead allow thieves to extract state directly from within stack frames of the producer. This is achieved by using state-map information provided by a cooperative runtime compiler, allowing us to drive down the cost of making state available for stealable work items. We discuss our design and preliminary findings for the implementation of our framework inside X10 work-stealing runtime and the optimizing compiler of Jikes RVM, a high-performance Java research virtual machine.

1. Introduction

Modern computer systems increasingly feature multicore computing units with deep memory hierarchies. Current object oriented languages such as Java are showing their age—particularly in the face of distribution, parallelism, and heterogeneity—all of which are rapidly becoming fundamental concerns. The new ubiquity of distribution, parallelism and heterogeneity has narrowed once distinct requirements, leading to the development of new languages such as X10 [7], Chapel [5], Fortress [1] and Cilk [8].

In modern parallel programming languages, the programmer exposes *potential* parallelism, leaving the runtime system responsible for scheduling tasks efficiently on the hardware. Scheduling algorithms based on Cilk's work-stealing scheduler provide dynamic, lightweight task parallelism, and are used by almost all modern parallel programming languages.

While work stealing has proven a valuable technique to exploit potential parallelism, it can come at a significant cost to the throughput of individual threads. This paper focuses on driving down this cost by reducing the overhead of ensuring all relevant state (such as that stored in global and local variables) is available to a potential thief at *every* steal point. For many applications, only a very small minority of potential steals is actually made, suggesting that where possible the cost of extracting state (from victim to thief) should be borne when a steal is made, and any unconditional penalties placed on the victim should be minimized.

The cost of making the state available is highly dependent on key implementation decisions. To our knowledge the implementers of modern parallel languages have taken the following implementation paths: 1) translation of application code into a low-level language such as C or C++ and compile with a runtime system implemented in C or C++; and/or 2) translation of application code into a high-level managed language such as Java. To implement work-stealing where the language is implemented by translation to C or C++, it is possible to compile a modified version of the code that maintains state within stack-allocated objects that may be copied to the heap at steal time. This is the approach taken by the existing C++ X10 work-stealing runtime. If the language is implemented by translation to a memory-safe high-level language such as Java, this approach is not possible, and so a straightforward implementation would instead revert to heap-allocating all stealable state. The question of what could be done along the lines of the stack-allocated object solution for C++ in the context of a modified Java virtual machine to support work stealing was the starting point for this work.

While the cost of allocating objects in Java may be significant, both of these implementation approaches can negatively affect the performance of application code because the code and data layout must both be restructured to allow for the possibility of work stealing, defeating various compiler optimizations. Since all relevant program state must be contained in either the hardware registers or the stack of the work-stealing victim, given the aid of a cooperative compiler it should be possible to map and then extract all state from the stack of the victim with only minimal changes to the executing code. This paper discusses preliminary work on the path to this goal.

Our implementation is based on the Java version of X10, and uses a modified version of Jikes RVM [2] to assist with extracting execution state. In this modified version a Java thread can force other running Java thread to yield and can copy relevant stack

frames from the other thread's Java stack onto its own stack. It can then walk on those stolen frames and retrieve the required execution states.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 provides relevant background on X10 and Jikes RVM. Section 5 explains the design of our Jikes RVM supported X10 work-stealing framework. Section 6 discuss the performance gain obtained over Fibonacci benchmark with our preliminary implementation. Finally, Section 7 concludes the paper.

2. Related Work

Load balancing is a challenging problem that has been well studied in the literature. Work-stealing has been used as a heuristic since Burton and Sleep's research [4] and Halstead's implementation of MultiLisp [11]. Blumofe et al. defined the fully-strict computation model and proposed a randomized work stealing scheduler with provable time and space bounds [3]. An implementation of this algorithm with compiler support for Cilk was presented in [8]. There have been several published algorithms since then for work-stealing, all adhering to the strong semantics. Fork-Join (FJ) is a java framework for recursively dividing the tasks into subtasks and is based on Cilk like work-stealing [12]. We have described the key difference between FJ and X10 work-stealing in section 3.3. Among recent work, Guo et al. implemented and evaluated work-first and help-first work-stealing policies [9]. However, their implementation saved execution state in heap-allocated frames. In later work, they also demonstrate an adaptive approach of choosing between work-first and help-first policies [10]. Zhao et al. presented the compiler transformation techniques for eliminating redundant task creation/termination and synchronization operations in task-parallel languages [10]. Wang et al. propose an adaptive task creation strategy for work-stealing that reduces the number of tasks created and provides a better control of the task granularities [13]. Instead of reducing the number of tasks created for effective work-stealing, our work attempts to reduce the cost of making work items available for stealing.

3. Background

In this section we will first briefly summarize the X10 language's finish-async parallel programming model, followed by the current implementation of the X10 work-stealing runtime with the help of *Frame* objects. We then briefly discuss about how the application states are extracted from its live stack frames.

The X10 work-stealing runtime described here is publicly available as part of X10, and to our knowledge the details of its implementation have not been published. This paper describes the work-stealing runtime available in the X10 Subversion head from release 20276 (X10 2.0). We do not claim the X10 work-stealing runtime as a contribution, but describe it in detail to assist with our explanation of the modifications made inside Jikes RVM to enable state extraction.

3.1 X10 finish-async Parallel Programming Model

X10 is a type-safe, modern, parallel, distributed object-oriented language developed by IBM as a part of the DARPA High Productivity Computing Systems (HPCS) program. It can be thought as an extension and modification of Java. X10 is generic, removes Java's concurrency, arrays and built-in types and adds places, activities, clocks, distributed multi-dimensional arrays and value types. Places make distribution explicit: each place can be thought as a virtual, architecturally homogeneous, shared memory multiprocessor.

X10 has asynchronous activities as a foundation for lightweight 'threads' that can be created locally or remotely. In X10 an asynchronous activity is created by a statement **async**`(P) S` where `P` is a place expression and `S` is a statement. The statement **async** `S` is treated as shorthand for **async**`(here) S`, where here is a constant that stands for the place at which the activity is executing [6] (X10 2.0). The execution of a statement by an activity is said to terminate locally when the activity has finished all the computation related to that statement and a statement is said to terminate globally when it has terminated locally and all activities that it may have spawned (if any) have, recursively, terminated globally. The statement **finish** `S` in X10 converts global termination to local termination. **finish** `S` terminates locally (and globally) when `S` terminates globally. If `S` terminates normally, then **finish** `S` terminates normally and `A` continues execution with the statement after **finish** `S`.

```
1  public static def fib(n:int) {
2    if (n<=2) return 1;
3    val f1:int;
4    val f2:int;
5    finish {
6      async { f1 = fib(n-1); }
7      f2 = fib(n-2);
8    }
9    return f1 + f2;
10 }
```

Figure 1. Sample X10 code

The activities in X10 can always launch another child activity. If there is the same governing finish for a parent and child activity then either of these parent and child activity can terminate in any order. This level of asynchrony is very useful in divide and conquer algorithms since this allows the X10 runtime to execute the sub-computations at different levels of the divide-and-conquer tree to execute in parallel without forcing synchronization at the parent-child level.

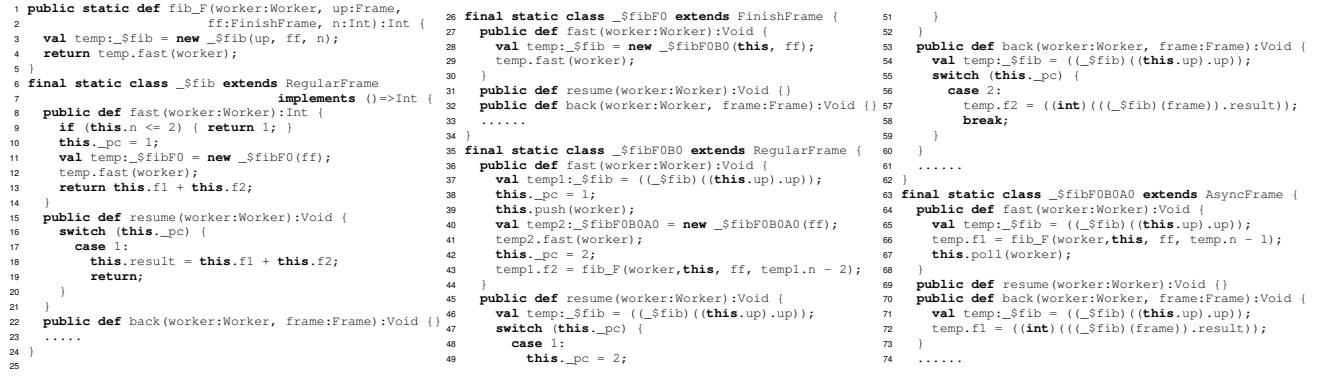
3.2 X10 Runtime Work-Stealing Scheduler

Although X10 is a distributed language, we are only dealing with the single *place* execution of X10 i.e. only over one node and hence, whenever we mention parallelism below, we mean at single node level.

In a work-stealing scheduler, the total number of worker threads is equal to the total number of available processors. The thread producing the tasks is called *victim* and the thread consuming the tasks i.e. stealing the tasks is called *thief*. Each worker thread has a double-ended queue, called as *deque* in which, they maintain the extra tasks to be stolen by the *thief*. All the points in the program where a *victim* pushes a task into its deque is called a *steal point* and the computation that the thief resumes after the steal is called as the *continuation* of the original computation.

X10 work stealing is composed of two stages. The first being the source to source code transformation (mostly to synthesize continuation for steal points) and a runtime. Whenever an activity is launched by an **async**, the statements following this **async** block are always left to be stolen by thief threads (assuming the **async** and the statement both have the same governing **finish**). In Figure 1 line 7 is a steal point. A thief can steal this continuation and can execute in parallel. But for any thief to steal the continuation, the execution states must be saved by the victim thread before executing line 6. For this, the victim saves its local execution states inside heap allocated class object, which also has a field for the program counter to help the thief resume the computation from the correct place.

Whenever a thread is idle, it becomes a thief and attempts to steal work from another thread's deque. The thieves always steal work from the top of other thread's deque. On the other



```
1  public static def fib_F(worker:Worker, up:Frame,
2                          ff:FinishFrame, n:Int):Int {
3    val temp:_$fib = new _$fib(up, ff, n);
4    return temp.fast(worker);
5  }
6  final static class _$fib extends RegularFrame
7                               implements ()=>Int {
8    public def fast(worker:Worker):Int {
9      if (this.n <= 2) { return 1; }
10     this._pc = 1;
11     val temp:_$fibF0 = new _$fibF0(ff);
12     temp.fast(worker);
13     return this.f1 + this.f2;
14   }
15   public def resume(worker:Worker):Void {
16     switch (this._pc) {
17       case 1:
18         this.result = this.f1 + this.f2;
19         return;
20     }
21   }
22   public def back(worker:Worker, frame:Frame):Void {}
23   .....
24 }
25
26 final static class _$fibF0 extends FinishFrame {
27   public def fast(worker:Worker):Void {
28     val temp:_$fib = new _$fibF0B0(this, ff);
29     temp.fast(worker);
30   }
31   public def resume(worker:Worker):Void {}
32   public def back(worker:Worker, frame:Frame):Void {}
33   ......
34 }
35 final static class _$fibF0B0 extends RegularFrame {
36   public def fast(worker:Worker):Void {
37     val temp1:_$fib = ((_$fib)((this.up).up));
38     this._pc = 1;
39     this.push(worker);
40     val temp2:_$fibF0B0A0 = new _$fibF0B0A0(ff);
41     temp2.fast(worker);
42     this._pc = 2;
43     temp1.f2 = fib_F(worker,this, ff, temp1.n - 2);
44   }
45   public def resume(worker:Worker):Void {
46     val temp:_$fib = ((_$fib)((this.up).up));
47     switch (this._pc) {
48       case 1:
49         this._pc = 2;
50
51     }
52   }
53   public def back(worker:Worker, frame:Frame):Void {
54     val temp:_$fib = ((_$fib)((this.up).up));
55     switch (this._pc) {
56       case 2:
57         temp.f2 = ((int)(((_$fib)(frame)).result));
58         break;
59     }
60   }
61   ......
62 }
63 final static class _$fibF0B0A0 extends AsyncFrame {
64   public def fast(worker:Worker):Void {
65     val temp:_$fib = ((_$fib)((this.up).up));
66     temp.f1 = fib_F(worker,this, ff, temp.n - 1);
67     this.poll(worker);
68   }
69   public def resume(worker:Worker):Void {}
70   public def back(worker:Worker, frame:Frame):Void {
71     val temp:_$fib = ((_$fib)((this.up).up));
72     temp.f1 = ((int)(((_$fib)(frame)).result));
73   }
74   ......
```

Figure 2. Pseudocode generated for X10 work-stealing runtime

hand, the threads push and pop their work from the bottom of their own deque. Assuming thread-1, which was idle, steals the object reference to the steal point at line 7 from thread-0's deque. Thread-1 starts with a copy of the computation at line 7 with modification at the beginning to restore the execution state i.e. with proper initialization of all local variables. Thread-0 is now a victim and thread-1 is its thief. When the victim returns after executing line 6, it first checks if the steal point that it pushed before the **async** at line 6 has been stolen. It finds that the frame was stolen. If by this time thread-1 has not yet completed the continuation then thread-0 becomes a thief itself, otherwise thread-0 will combine its calculation with that from thread-1 and will complete the computation from line 8.

This method of executing the **async** before executing the continuation following the **async** block, is called as the *work-first* policy since a thread goes on to do its work (in an **async**) first, as in the case of serial execution [9]. The current work-stealing scheduler in X10 is based on this work-first policy.

3.3 Comparison to the Fork-Join Framework

The Fork-Join framework (FJ) [12] is a Java framework in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel, waiting for them to complete, and then composing results [12]. This methodology is very similar to what X10 tasks tries to achieve. However, in FJ, the user has to write his code such that it has the calls to the work-stealing framework, but in X10 the code generation phase automatically take care of this. The user simply expresses the parallelism in terms of *async* and *finish* block and the X10 runtime calls to work-stealing scheduler is inserted during the codegen pass. This also means the user does not have to worry about task dependencies in X10. FJ has a granularity parameter that represents the point at which the overhead of generating tasks outweighs potential parallelism benefits. This is currently not implemented inside X10 work-stealing runtime.

3.4 X10 Frame Objects

The X10 work-stealing runtime creates a frame class for each program scope. Each frame class contains the following - *fast* method for normal execution, *resume* method for continuing the execution after the steal (at given PC), *back* method for returning from callee (at given PC), pointer to the parent frame (*up* field), fields for local variables and result field if non-void method. The frames are of three types - Regular frame, Async frame and Finish frame.

Figure 2 shows how the X10 work-stealing runtime expands the input source code in figure 1 in terms of the frame classes. The *fast* method represents the fast path which a thread executes until its frames are stolen. The steal points are always available in the *RegularFrame* under the finish scope and are always pushed by the worker thread in this frame. Before this worker thread returns from the *AsyncFrame*, it always *poll* to check if the continuation was stolen. If it was stolen it first checks if the lost continuation is already finished by the thief. It does this by decreasing the async count of the *FinishFrame*, which was incremented by the thief before executing this continuation. If the async count is zero, the original worker thread starts unwinding the frames by calling the *back* method of *AsyncFrame*. However, if the async count is not zero, this worker thread becomes a thief.

4. Jikes RVM Extension For State Stealing

This section discusses the modifications made inside the Jikes RVM to allow a Java thread to steal states from another Java thread. We have done our modification inside Jikes RVM from subversion head with release 16041. We have a working implementation for both baseline and opt compiled frames, but the implementation is not fully optimized. In the process of a steal in our system, a thief:

- Forces a victim to yield.
- Copies victim stack frames onto its own Java stack.
- Walks over frames from where the victim would have returned.
- Saves relevant state into a linked list and unwinds the stack.
- Releases the victim to continue its execution.

To achieve this, we use the Jikes RVM yieldpoint mechanism (which is used for several key functions, including adaptive sampling, garbage collection safe-pointing, and biased locking bias revocation). We also amend the work-stealing application code to include frame pointers for queued work items—rather than pointers to state objects used in the default system—to allow thieves to discover the stack frames from which to extract state.

5. Modifications to X10 Runtime

To support work-stealing, the default X10 runtime performs a two stage transformation, first an X10 to X10 transformation to target the work-stealing runtime, and then an X10 to Java transformation (X10 also supports a C++ backend, motivating this two-stage process). Figure 2 shows the expanded code for the Fib.x10 application as shown in Figure 1 for the unmodified work-stealing runtime.

For our Jikes RVM assisted work-stealing runtime, we modify the original code (as in Figure 2) to take advantage of the stack state extraction available in our framework. An extract of the modified code is shown in Figure 3.

5.1 Fast Path Execution of Workers

One object for each of the *frame* classes used is preallocated. There is always a *RegularFrame* class, which contains all the variables on which the asyncs operate in each of the parallel sections. An array of such class object is also preallocated with the array size as total number of runtime X10 threads. An object at slot *n* is private to a

```
1  public static _$fib _$fib_obj=new _$fib();
2  public static _$fib[] threadLocalParam = new _$fib[THREADS];
3  public static _$fibF0 _$fibF0_obj=new _$fibF0();
4  public static _$fibF0B0 _$fibF0B0_obj=new _$fibF0B0(2);
5  public static _$fibF0B0A0 _$fibF0B0A0_obj=new _$fibF0B0A0();
6  ......
7  final static class _$fibF0B0 extends RegularFrame {
8    public def fast(threadLocalParamObj:RegularFrame):Int{
9      int _pc = 1;
10     int n = threadLocalParamObj.n;
11     worker.push(framePointerToThisMethod);
12     f1 = _$fibF0B0A0_obj.fast(threadLocalParamObj);
13     if(rvmAssistedStealInProgress) {
14       /*Thief Saves: _pc,framePointerToThisMethod,n*/
15       return -1;
16     }
17     _pc = 2;
18     n = n -2;
19     threadLocalParamObj.n = n;
20     f2 = fib_F(threadLocalParamObj);
21     if(rvmAssistedStealInProgress) {
22       /*Thief Saves: _pc,framePointerToThisMethod,n,f1*/
23       return -1;
24     }
25     return f1+f2;
26   }
27 public def resume(worker:Worker,_pc:Int,obj:Object){
28   switch (_pc) {
29     case 1:
30       _pc = 2;
31       threadLocalParamObj = threadLocalParam[THREAD_INDEX];
32       worker.saveAsHighestFrame(framePointerToThisMethod);
33       int n = getStateValue();
34       threadLocalParamObj.n = n - 2;
35       this.updatePartialResultNodeWithMyFPAddress();
36       ((_$fib)(obj)).f2 = fib_F(threadLocalParamObj);
37       if(rvmAssistedStealInProgress) {
38         /*Thief Saves: _pc,framePointerToThisMethod*/
39         return;
40       }
41   }
42 }
43 public def back(worker:Worker,_pc:Int,obj:Object){
44   switch (_pc) {
45     case 2:
46       ((_$fib)obj).f2 = ((_$fib)obj).result;
47       ((Fib._$fib)obj).f1 = getStateValue();
48       break;
49   }
50 }
51 ......
52 }
53 final static class _$fibF0B0A0 extends AsyncFrame {
54   public def fast(threadLocalParamObj:RegularFrame):Int{
55     threadLocalParamObj.n = threadLocalParamObj.n - 1;
56     f1 = fib_F(threadLocalParamObj);
57     if(rvmAssistedStealInProgress) {
58       /*Thief Saves: _pc,framePointerToThisMethod*/
59       return -1;
60     }
61     boolean frameStolen = worker.poll();
62     if(frameStolen) {
63       _$fib t = new _$fib();
64       t.f1 = f1;
65       worker.stolen(t);
66     }
67     return f1;
68   }
69   public def resume(worker:Worker,_pc:Int,obj:Object){}
70   public def back(worker:Worker,_pc:Int,obj:Object){
71     ((_$fib)obj).f1 = ((_$fib)obj).result;
72   }
73   ......
74 }
```

Figure 3. Pseudocode generated for Jikes RVM supported X10 work-stealing runtime

worker thread with thread index *n*. This index is assigned by X10 runtime and is always between 0 and (total_threads-1). To avoid passing all the required states as parameters to function, we use this object instead. All of the above preallocations are as on lines 1-5 in figure 3. The methods at lines 8 & 54 denote the fast path. The *RegularFrame* _$fibF0B0 contains a *push* method (provided by the default implementation of X10 work-stealing runtime). The total number of steal points in this class is equal to the total number of calls to *push*. So the total number of parallel sections inside this class is the total number of *push* calls plus 1. This is used by the modified runtime to set up its internal data structure, which helps in deciding global termination across all of the parallel sections. The runtime is informed about the total number of parallel sections at line 4. The *push* method is modified to pass the pointer to this current frame (on its Java stack). This frame pointer is found by existing mechanisms inside Jikes RVM. With this modification now each worker thread's deque stores this frame pointer instead of the reference to the object containing all the required states. The *fast* method of every frame is then modified to pass the preallocated object (at line 2) as parameter to the *fast* method of the next class. As with default X10 work-stealing runtime, when a worker thread enters the *fast* method containing parallel sections, it first performs a *push* and then carries out the computation so the continuation can be computed in parallel. However, there is one significant difference from the default implementation. In default work-stealing mode, as shown at lines 3, 11, 27 and 39, in figure 2, before calling the *fast* method of next class, it was required to do an object allocation of next class and then call the *fast* method with this object. This object of next class contains a reference to the earlier class. This was done to assist a worker thread in computing the slow path, where it has to gather the partial computations from each worker thread i.e. to know the local states at each class. But with our implementation these object allocations are not required at all.

Once the activity launched by the async is finished, the worker thread *poll* and check if the previously left continuation (by the last *push*) have been stolen. However, in our implementation, if they have been stolen, the worker allocates an object and stores its partial calculation result inside it. This object is then handed over to the runtime. After this, the worker thread continues to follow the default X10 work-stealing execution path i.e. the thief starts searching for a continuation to steal and for the case where the previous call to *poll* found that the continuation was not stolen, it simply removes the corresponding frame entry from its deque and start unwinding its frames.

5.2 Stealing and Executing the Continuation

As in the original work-stealing runtime, a thief worker thread continues to search for continuations to steal in the deque of all other worker threads. With our implementation, when it steals the frame pointer from the deque of the victim worker thread, it initiates the Jikes RVM state extractor with two input parameters - the frame pointer stolen from the deque (say *frame_A*) and the frame pointer of the first executed application method (say *frame_B*). The *frame_B* is on the higher memory side of the victim's Java stack and the *frame_A* is on the lower memory side of victim's Java stack. The method corresponding to *frame_B* is the first application method executed by the victim. Thief extract the states from each of the frames starting *frame_A* of the victim to *frame_B* as explained in the section 4 above. For the modified code in figure 3, once a thief starts the application after extracting the states from victim's stack frames, it sets its own *frame_B* as the first application method, which is *resume* method of *_$fibF0B0* class, as shown in line 32. Once inside the *resume* method the thief initializes the required states as shown at line 33 in figure 3. It then update the runtime with the stack frame pointer of its current invocation of the _$fibF0B0.*resume* method as at line 35. This follows the call to _$fibF0B0.*fast* at line 36, which marks the start of the thief's fast path execution.

5.3 Global Termination Data-structure (GTD)

In the default X10 work-stealing runtime, a global termination over a *finish* block is determined by decrementing the total *async* count (whenever an *async* returns after computing the continuation) at each *FinishFrame* object (corresponding to this *finish* block), which was incremented when an *async* started executing the continuation stolen from this *finish* scope. When this count becomes zero, a global termination is performed.

However, in our implementation, we have only one object allocated for a *FinishFrame* class, unlike one object for each *finish* block as in case of default X10 work-stealing runtime, and hence we need some other mechanism to determine a global termination over each *finish* blocks. For this, we maintain a data structure, which is in the form of a linked list[1] and is maintained by the modified X10 work-stealing runtime. The information held at each node of the list is:

- The linked list of states retrieved by the RVM from the victim.
- A list of frame pointers containing - address of stolen frame from the victim and the address of *resume* frame from where the thief started executing the continuation.
- Field indicating total parallel sections available in the *RegularFrame* class of the stolen *fast* frame from the victim.

[1] Although we are in the process of converting this to a tree data structure in our final implementation.

Time	Worker	Work Done	Last Steal GTD Node Index	Current GTD Node Index
T0	A	Starts Application	-	0
T0	B	Waiting to Steal	-	0
T0	C	Waiting to Steal	-	0
T1	A	Push: C0	-	0
T2	A	Push: C1	-	0
T3	B	Steal: C0	-	0
T4	B	Add(GTD_Node0)	**Node0: 0;**	**0**
T5	A	Updated by B	Node0: 0;	**0**
T6	C	Steal: C1	Node0: 0;	0
T7	C	Add(GTD_Node1)	Node0: 0; **Node1: 0;**	**1**
T8	A	Updated by C	Node0: 0; Node1: 0;	**1**
T9	B	Push: C2	Node0: 0; Node1: 0;	0
T10	C	Steal: C2	Node0: 0; Node1: 0;	1
T11	C	Add(GTD_Node2)	Node0: 0; Node1: 0; **Node2: 0**	**2**
T12	B	Updated by C	Node0: 0; Node1: 0; Node2: 0	**2**

Figure 4. Updation of *last_steal_GTD_node_index* & *curr_GTD_node_index* fields.

- The partial results computed by the worker thread when they realize that they cannot continue the remaining computation as it was stolen or if a global termination is still pending.
- The total number of registrations completed by a worker thread for this node of the data structure. A worker thread increments this registration field whenever it returns back after performing the computation. A victim increments this when it finds the continuation was stolen and a thief increments it when it has finished computing the stolen continuation.
- Integer field - *last_steal_GTD_node_index* - indicating the index of the parent node in the data structure. Each worker has an integer field - *current_GTD_node_index* - which is initially zero. The update procedure for these fields is explained by Figure 4.

Before creating a new node inside GTD, the runtime always checks if there is already a node registered with the currently stolen frame pointer. For doing this, it starts scanning backwards from the last node inside the GTD. For every node it checks if the stolen frame address is already stored inside the list of frame pointers at the GTD node. If the address is already registered then it does not create a new node.

5.4 Stack Unwinding

If none of the frames of a worker thread are stolen, unwinding equates to the normal unwinding of the Java stack. However, unwinding after a steal requires runtime assistance. When a worker thread returns to a steal point after completing the computation, the runtime increments the total registration counts for the steal's GTD node. If the total registrations is not same as the total number of parallel sections then it does not allow this worker thread to unwind—this worker thread will become a thief. When a worker thread returns after completing a computation, it can then allocate an object and store its partial calculation result inside it, which is then attached to the GTD node corresponding to the steal. If there is already a partial result attached to this GTD node then the computation is updated within that object.

To keep track of which is the parent frame of the current frame, each worker thread has to scan the list of states saved inside the GTD node. The state retrieved by the thief is a linked list node. Each node corresponds to one state and contains the frame pointer whose state is stored in this node. For every new stack frame, the first node will contain the reference to the class whose method's stack frame is being read. By using this reference the worker thread can call the *resume* or *back* methods of that class. As with default implementation, if the worker thread is a thief then it starts unwinding by calling the *back* method of the *RegularFrame* class inside which the steal was performed. In our implementation, it can however get the local states at each *back* or *resume* method by reading the state linked list. If the worker thread is the victim then it will walk the state linked list until it reaches the next frame. Now as in default X10 work-stealing runtime, both thief and victim unwind by first calling the *back* method and then the *resume* method for each class until it reaches the bottommost application stack frame, or it encounters a frame from *RegularFrame* class for which the global termination is still pending. In this case the thread becomes a thief and starts hunting for frames to steal. In our implementation, each of the *back* method and the *resume* method passes the computed result to the runtime, which then passes it to the invocation of next class's *back* method and then the *resume* method.

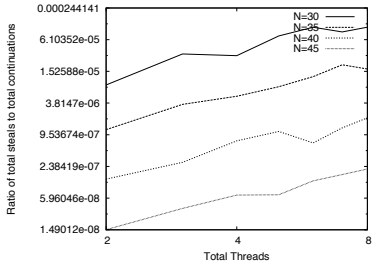


Figure 5. Ratio of steals to total continuations.

While unwinding, a worker thread always needs to check if there is any pending global termination for the frame it is currently processing. As mentioned above, the unwinding is always started from a steal point. In our implementation this has a corresponding GTD node. The field *last_steal_GTD_node_index* in this node gives the index of its parent node. For any X10 application the steal points are only inside the *RegularFrame* class containing *push* method. While unwinding, whenever the worker thread encounters any frame from such a *RegularFrame* class it queries the runtime if any continuation was stolen here. To achieve this the runtime compares the frame pointer of this frame with the frame pointers stored at the parent GTD node. If they match then the worker has hit another steal point while unwinding. The runtime then increments the pending registration counts and then checks if it is equal to the total number of parallel sections. If so, it allows the worker to continue the unwinding. When worker starts unwinding from this frame then it starts by reading states from the list stored at this GTD node, discarding the earlier state list.

It is very important that the runtime compares the frame pointer from *RegularFrame* classes, containing parallel sections, only with the frame pointers stored at the parent GTD node. Whenever a worker thread becomes a thief, it starts with a stack frames with zero X10 application frames. Due to this it is possible that the stack frame pointer for a method (e.g. *_$fibF0B0.fast*) invoked from steal-A is same as that of its invocation from steal-B by the same worker thread. However, until a victim becomes a thief, it is guaranteed that every frame on its stack has a unique address.

6. Results

All our measurements use the Fibonacci benchmark. While this is not ideal, we do not have X10 versions of all the regular benchmarks which we can test over our implementation. We hope to improve this situation in the future.

6.1 Fraction of Continuations Stolen

We performed all experiments with the production build of Jikes RVM and the machine we used was one node of Core i7 920 Bloomfield (Family 6 Model 26). In every run, the total number

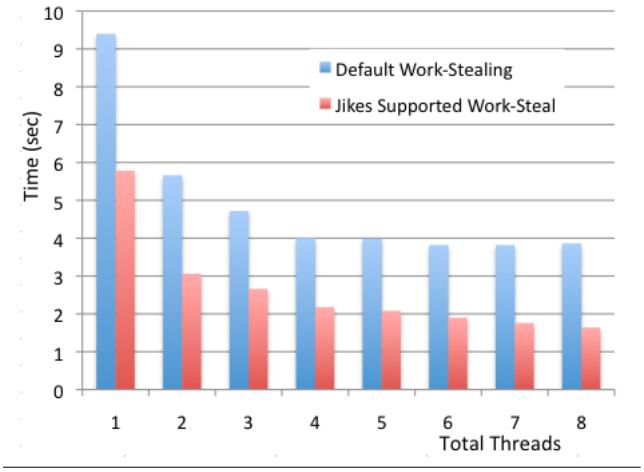


Figure 6. Execution time for work-stealing implementations.

of garbage collector threads were equal to the total number of X10 worker threads.

To verify our belief that few continuations are stolen in practice, we ran Fib.x10 using the original implementation of the X10 work-stealing runtime for several problem sizes and a range of worker thread counts. We calculated the total number of steals in each run, as well as the total number of continuations produced by all threads, and report the ratio of stolen to total continuations in Figure 5. This experiment shows that (at least for this benchmark) only a very small fraction of continuations are actually stolen. In such situations it is clear that any additional effort made to eagerly make available local states at potential steal points would be largely wasted. Other benchmarks, such as LU, have been shown to exhibit a steal ratio as high as 20% [12], and we intend to implement such benchmarks to evaluate our system in the future.

6.2 Performance

Figure 6 compares the execution time of Fibonacci (for N=40) with our implementaton versus the default work-stealing implementation in X10 (using the Java back-end). Initial few iterations during the benchmark execution is discarded considering the warmup time and the execution time mentioned in the graph is the average of last five iterations. The graph shows that the execution time with our system is significantly lower than the default X10 work-stealing implementation, which is a promising result. With different problem sizes, we have found that the execution time with our implementation is around 1.5x to 3.3x faster than the default Java X10 work-stealing, with the difference increasing for larger problem sizes.

7. Conclusion

In this paper we described the design of a work-stealing framework within a Java virtual machine that provides an alternative to explicitly saving execution states whenever a stealable work item is created. We measured that in several applications, the vast majority of work items are never stolen, but are instead processed by the producer itself. Our design allows the thief to retrieve execution state directly from the stack frame of the victim when a steal is performed. We describe our implementation within the X10 work-stealing runtime and Jikes RVM. We even calculated the performance of our implementation over Fibonacci benchmark. Although we have promising results, our implementation still has to include many further optimizations. However, with these preliminary results we are highly excited and we anticipate that our final implementation will have significant performance benefits over the conventional way.

8. Acknowledgments

The authors would like to thank all of the X10 team for their work on the language and implementation.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

[2] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2010. ISSN 0018-8670.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999. ISSN 0004-5411.

[4] F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194. ACM, 1981. ISBN 0897910605.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291, 2007. ISSN 1094-3420.

[6] 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 annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005. ISBN 1595930310.

[7] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proc. of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, pages 45–52. Citeseer, 2005.

[8] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998. ISSN 0362-1340.

[9] 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*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[11] R. Halstead Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17. ACM, 1984. ISBN 0897911423.

[12] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000. ISBN 1581132883.

[13] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P. Yew. An adaptive task creation strategy for work-stealing scheduling. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 266–277. ACM, 2010.