Compiler Support for Work-Stealing Parallel Runtime Systems

Raghavan Raman, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar

Department of Computer Science, Rice University
{raghav,jisheng.zhao,zoran,vsarkar}@rice.edu

Abstract. Multiple programming models are emerging to address an increased need for dynamic task parallelism in multicore shared-memory multiprocessors. Examples include OpenMP 3.0, Java Concurrency Utilities, Microsoft Task Parallel Library, Intel Threading Building Blocks, Cilk, X10, Chapel, and Fortress. Scheduling algorithms based on work-stealing, as embodied in Cilk's implementation of dynamic *spawn-sync* parallelism, are gaining in popularity but also have inherent limitations. In this paper, we focus on the compiler support needed to extend work-stealing for dynamic *async-finish* task parallelism as supported by X10 and Habanero-Java (HJ). We discuss the compiler support needed for work-stealing with both the *work-first* and *help-first* policies. Performance results obtained using our compiler and the HJ work-stealing runtime show significant improvement compared to the earlier work-sharing runtime from X10 v1.5.
We also propose and implement three optimizations, *Dynamic-Loop-Chunking*, *Redundant-Frame-Store*, and *Objects-As-Frames*, that can be performed in the compiler to improve the code generated for work-stealing schedulers. Performance results show that the Dynamic-Loop-Chunking optimization significantly improves the performance of loop based benchmarks using work-stealing schedulers with work-first policy. The Redundant-Frame-Store optimizations provide a significant reduction in the code size. The results also show that our novel Objects-As-Frames optimization yields performance improvement in many cases. To the best of our knowledge, this is the first implementation of compiler support for work-stealing schedulers for async-finish parallelism with both the work-first and help-first policies and support for task migration.

1 Introduction

The computer industry is entering a new era of mainstream parallel processing due to current hardware trends and power efficiency limits. Now that all computers — embedded, mainstream, and high-end — are being built using multicore chips, the need for improved productivity in parallel programming has taken on a new urgency. The three programming languages developed as part of the DARPA HPCS program (Chapel [8], Fortress [2] and X10 [9]) all identified dynamic lightweight task parallelism as one of the prerequisites for success. Dynamic task parallelism is also being included for mainstream use in many new programming models for multicore processors and shared-memory parallelism, such as OpenMP 3.0 [5], Java Concurrency Utilities [19], Microsoft Task Parallel Library [13], Intel Threading Building Blocks [20]

and Cilk [16]. In addition, dynamic data-driven execution has been identified as an important trend for future multicore software [14], in contrast to past programming models based on the Bulk Synchronous Parallel (BSP) and Single Program Multiple Data (SPMD) paradigms.

A standard approach for supporting dynamic parallelism can be found in *work-sharing* runtime systems, in which dynamic tasks are stored in a *shared queue* and *worker threads* repeatedly pick new tasks for execution. The JUC Executor framework described in [19] is an example of a work-sharing runtime system. A major advantage of the work-sharing approach is that it can be implemented as a library without requiring any compiler support. However, there are two major limitations in the work-sharing approach. First, the shared queue often becomes a scalability bottleneck. Second, whenever a task performs a blocking operation, its worker thread needs to suspend as well, thereby forcing the runtime system to create more worker threads to ensure progress. An implementation of the X10 v1.5 runtime system based on work-sharing for symmetric shared-memory multiprocessors (SMPs) is described in [3]. It uses X10ThreadPoolExecutor which is an extension of the JUC ThreadPoolExecutor.

Scheduling techniques based on *work-stealing* aim to overcome these limitations of work-sharing runtimes. First, they only maintain a *local* queue (deque) per worker thread to improve scalability. Second, they include compiler and runtime support for storing task's suspended state on the heap so that its worker thread can switch to another task instead of creating a new worker thread. As described in [16] for Cilk's spawn-sync parallelism, these techniques require compiler support to ensure that local variables of a task can be saved and restored within different workers. This paper addresses the challenges involved in compiler support for work-stealing schedulers for more general async-finish parallel constructs [17].

This paper focuses on a core subset of the concurrency constructs of the HabaneroJava (HJ) language, which is an extension of X10 $v1.5$, consisting of the async, atomic, and finish constructs, as well as a new *delayed async* construct [6]. We describe the compiler support needed to extend work-stealing to HJ's dynamic async-finish task parallelism, which is more general than Cilk's spawn-sync parallelism. We also discuss the compiler support needed for work-stealing using both the *work-first* and *help-first* policies. We explore some opportunities for compile-time code optimizations for these constructs. Though the code generated by the compiler targets the Habanero Work-Stealing Runtime System [17], the techniques described in this paper are general and can be used to target any work-stealing runtime system. The main contributions of this paper are:

- We extend past work on compilation techniques for spawn-sync work-stealing schedulers to also support async-finish parallelism
- We describe the changes needed in the compiler to support work-stealing scheduling under the help-first policy.
- We describe the dynamic loop chunking optimization which reduces the number of steals in parallel loops when work-stealing with the work-first policy is used.
- We propose and implement two data-flow analyses for removing redundant frame-store statements that are inserted by the compiler.
- We introduce a new optimization that uses application objects as *TaskFrames*, under work-stealing with the help-first policy.

2 Work-Stealing Extensions

Our work targets the HabaneroJava (HJ) programming language and model [21] that implements the Java-based X10 v1.5 language [9] on multicore processors, along with several programming model extensions. In our past work [17], we extended the work-stealing technique described in [4], which was designed and implemented for the Cilk language, to a broader class of computations that are generated by HJ's finish-async constructs. This section summarizes a variety of issues that arise due to these constructs and motivates the need for new compiler and runtime extensions to handle these issues.

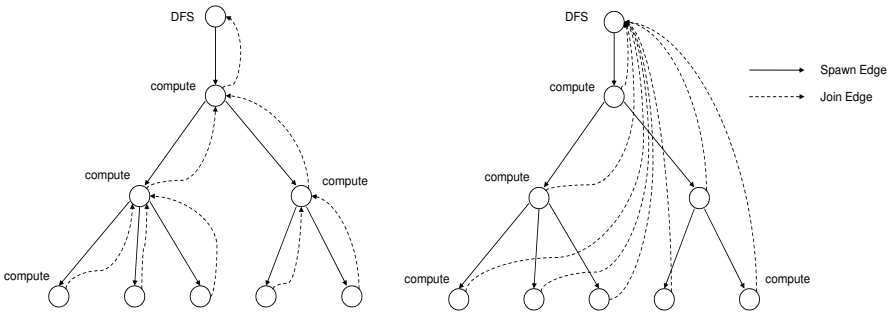
The async statement in HJ creates a new task that can be executed in parallel with its parent. The finish statement ensures that all the tasks that were created in the scope of the finish complete before proceeding to execute the next statement following the finish. HJ's async and finish statements are analogous to Cilk's spawn and sync constructs, with some important differences discussed below.

```
 1 class V  {
 2   V [] nbors;
 3   V parent;
 4   V (int i) {super(i); }
 5   void compute() {
 6     for (int i=0;i<nbors.length;
           i++) {
 7       V e = nbors[i];
 8       if (e.tryLabeling(this))
           // escaping async
 9         async e.compute();
10     }}
11   boolean tryLabeling(V n) {
12     atomic if (parent == null)
13               parent = n;
14     return parent == n;
15   }
16   void DFS() {
17     parent = this;
18     finish compute();
19   }}
```

Fig. 1. Code for parallel DFS spanning tree algorithm in HJ

Escaping Asyncs. In HJ, it is possible for a descendant task to outlive its parent. For instance, consider the parallel DFS spanning tree graph algorithm [11] whose HJ code is shown in Figure 1. A single finish scope at line 18 suffices for all descendant tasks spawned at line 9. The spawned tasks can still be alive after its parent (*compute*) returns — we refer to such tasks as *Escaping Asyncs*. In contrast, in Cilk there is an implicit sync operation at the end of each function, ensuring that every parent waits for all its children to complete. Semantically, this would be equivalent to adding a finish scope to enclose the body of the *compute* function in HJ. Figure 2 shows the spawn tree of Cilk's *fully strict* [4] and HJ's *terminally strict* [1] computation for the program in Figure 1. Escaping asyncs demand additional compiler and runtime support for work-stealing.

Sequential Calls to Parallel Functions. Parallel functions are defined as those that may spawn parallel tasks either directly or indirectly by calling other parallel functions. In Cilk, sequential calls to parallel functions (known as *cilk functions*) are not allowed, thus all functions that may appear on a call path to a spawn operation must be designated as cilk functions and must also be spawned. This is a significant restriction for the programmer because it complicates converting sequential code to parallel code, and prohibits the insertion of sequential code wrappers for parallel code. In contrast, HJ permits the same function to be invoked sequentially or via an async at different program points. This imposes a challenge for work-stealing scheduling as it introduces new continuation points located immediately after sequential calls to parallel functions.



(a) Cilk (b) HJ

Fig. 2. Spawn tree for DFS spanning tree algorithm for Cilk and HJ

Arbitrarily Nested Asyncs. The syntax of spawn in Cilk requires that a spawned task must always be a procedure call. This has a significant software engineering impact since every piece of code that needs to be executed in parallel has to be wrapped in a procedure. In contrast, the async construct in HJ allows parallel task creation for any arbitrary piece of code. This arbitrary nesting of asyncs in HJ presents another challenge for work-stealing scheduling because there may be more than one worker executing different parts of the same execution instance of a function at the same time, and the data for these tasks need to be stored in a manner that ensures no data races among nested or parallel tasks in the same function.

Work-First and Help-First Policies. According to Cilk's work-stealing strategy [16], a worker executes a spawned task and leaves the continuation to be stolen by another worker. We call this a *work-first* policy. In [17], we proposed an alternate strategy to steal work, called a *help-first* policy, in which the worker executes the continuation and leaves the spawned task to be stolen. This approach is better suited for certain kinds of applications [17]. The term "help-first" suggests that the worker will ask for help from its peer workers before working on the task itself. The compilation support for scheduling under help-first policy is detailed in Section 3.2.

Need for Compiler Support. While the work-stealing runtime can be implemented as a library and exposed to the programmer, this would put severe burden on the programmers since they must now know the details of the runtime library and must also complete a non-trivial task of managing the state for task suspend and resume operations. Instead, we propose higher level abstractions in the programming language to represent parallelism, with compiler support to transform these abstractions to use the work-stealing library. This helps in improving productivity since the programmers now do not need to worry about the details of the implementation of parallelism.

3 Compiler Support for Work-Stealing Extensions

This section describes the compiler support needed for the work-stealing extensions. The code generation described here is performed in the Habanero Parallel Intermediate Representation (PIR), which is an extension of the Soot Java optimization framework [24], but can be done in any reasonable optimizing compiler framework.

3.1 Support for Extended Work-Stealing with the Work-First Policy

We define *parallel* functions as those that contain either a finish or an async. These are the constructs that directly result in a continuation which might be executed by a thief. In order to collect a list of parallel functions, the compiler needs to scan all function bodies to look for an async or a finish.

A function is *potentially parallel* if it contains a call to a parallel or a potentially parallel function. We perform a conservative static inter-procedural analysis to identify the potentially parallel functions. First, we build the call-graph of the entire application program using the Class Hierarchy Analysis [12, 15] from the Soot framework [24].

Once we have the call-graph, we use a Depth-First-Search (DFS) on the reverse call-graph to mark all functions reachable from a parallel function as potentially parallel. This step is linear in the number of edges in the call-graph of the application. The step to identify the parallel functions is a simple linear scan of the code that marks all functions that contain finish or async as parallel.

Sequential Calls to Potentially Parallel Functions. In Cilk [16], each cilk function corresponds to a task in the computation dag. A parallel function cannot be called sequentially and must be spawned. This simplifies the task for the work-stealing scheduler because the continuation only contains the activation frame of the current function since the thief that executes the continuation will never return to its caller as the caller is the parent task whose continuation was already stolen. In contrast, sequential calls to potentially parallel functions are allowed in HJ, resulting in a continuation point immediately after the call. Below are three approaches for supporting such sequential calls.

A naïve way to handle sequential calls to potentially parallel functions would be to wrap the call in a finish-async construct. The problem with this approach is that it loses parallelism by disallowing the code after the sequential call to run in parallel with the task that escapes the callee of the sequential call.

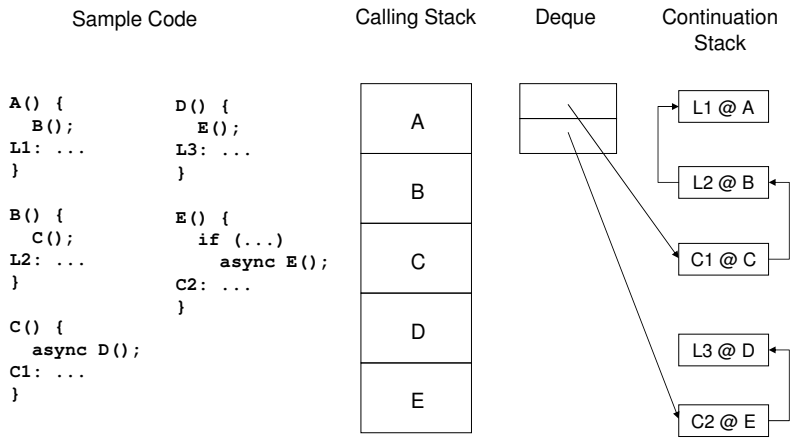


Fig. 3. Supporting Sequential calls to Potentially Parallel Functions by maintaining the call stack on heap frame

The sequential calls to potentially parallel functions have a property that the continuations after such calls can start immediately after the callee returns. One way to model this is to allow the worker that completes the callee to continue executing the continuation in the caller, thus doing some more useful work without stealing. In order to make this possible, we extend every continuation to contain a stack of activation frames up to the previous async in the call chain. When compiling a sequential call to a potentially parallel function, we do not put the continuation on to the stealing deque, but instead we push it onto the stack of continuations from the previous async in the call chain.

Consider the example in Figure 3. $C1$ and $C2$ labels denote the points where stealing can actually occur. At $C1$, the frame pushed on to the deque contains the stack of activation frames for $C1$, $L2$, $L1$ in that order. The thief that steals the frame is responsible for starting the continuation at $C1$. Upon returning from function C, the thief will resume the execution at $L2$, after which it will return to $L1$. At each return, the thief will find the activation frame required to resume the execution by popping the stack. Similarly, the activation frame stack for $C2$ contains the activation frames up to the previous spawn i.e., $C2$ and $L3$. This approach allows the continuation after the sequential call to execute in parallel with the task that escapes the callee.

```
   A() {
 1   S0;
 2   finish {//startFinish
 3     async {
 4       S1;
 5       async { S2; }
 6       S3;
 7       async { S4; }
 8     }
 9     S5;
10   }//stopFinish
11   S6;
   }
```

```
   A() {
     frame1 = new AFrame;
 1   S0;
 2   finish {//startFinish
 3     async {
         frame2 = new async1Frame;
 4       S1;
 5       async { S2; }
 6       S3;
 7       async { S4; }
 8     }
 9     S5;
10   }//stopFinish
11   S6;
   }
```

Fig. 4. HJ code to depict the use of *frames* in the case of arbitrarily nested asyncs.

Arbitrarily Nested Asyncs. In HJ, an async can contain any arbitrary statement, allowing asyncs with arbitrary nesting depths. With no constraint on the nesting of asyncs, there can be more than one worker executing the same instance of a function, i.e., one worker can be executing the parent task while the other executes its child task. Thus having one frame for every function instance is no longer sufficient; we need one heap frame per task/activity in order to allow the child tasks in the nested asyncs to run in parallel with their parents. Figure 4 shows an HJ code fragment that contains arbitrarily nested asyncs and an equivalent version with frames created at appropriate points. Note that there are two frames that are being created in this method, one for the parent task executing the method and the other for the child task executing the async, since both tasks contain continuations.

3.2 Compilation Support for the Help-First Policy

In help-first policy, the worker that creates the tasks defined by asyncs does not execute these tasks immediately, but instead makes the new tasks available to other workers for stealing and proceeds to execute the continuation after the async. This is in contrast to the work-first policy implemented in Cilk [16], and requires different code generation. This section summarizes the code generation needed for help-first work-stealing.

In help-first policy, we use a different definition of a *parallel function* to reflect the fact that asyncs no longer create continuation points, even though they are transformed to create tasks available for stealing. Only the termination of a finish construct results in a continuation point for the help-first policy. Hence, we now define a *parallel function* as one that contains a finish construct. *Potentially parallel* functions are defined as before, as functions that are either parallel or call other potentially parallel functions.

Task Frames for Asyncs. Help-first policy dictates that tasks defined by async's are made available to idle workers for stealing. The compiler needs to wrap them in a data structure holding all the information needed to execute the async. This includes values of all local variables used in the async, the body of the async and the *Immediately Enclosing Finish* that this task reports to on completion. We use a *TaskFrame* data structure to store this information; one has to be created for every instance of an async.

We also need a data structure similar to the one used in work-first scheduling to store the activation frame of the function. We call this a *ContinuationFrame*. The *ContinuationFrame* holds the local variables used in the continuation of the function, and the program counter indicating the continuation point at which the execution resumes. It can also hold a reference to the *ContinuationFrame* of the parent function if this function is called sequentially. There is no need for the *ContinuationFrame* for functions that do not contain any continuation points.

4 Optimizations in Work-Stealing Compilation

Compilation for work-stealing parallel runtime systems transforms parallel constructs such as finish and async into code that implements the parallel semantics of these constructs on top of the underlying runtime system. This section describes some optimizations that can be performed on the code transformed for a work-stealing parallel runtime. Section 4.1 describes an optimization to improve the performance of the work-first policy based work-stealing on parallel loops. Section 4.2 discusses the problem of identifying redundant stores into the heap frame data structure. Section 4.3 introduces a new optimization that identifies application objects that can be used as *TaskFrames* under work-stealing with the help-first policy, thereby eliminating the overhead of allocating *TaskFrames*.

4.1 Dynamic Loop Chunking Optimization

The work-stealing algorithm using the work-first policy performs very well on recursive divide-and-conquer applications, because the number of steals for such applications is usually very low. Unfortunately, when the parallelism in the application is expressed in

the form of parallel loops[1], the performance of work-stealing with the work-first policy suffers. The dynamic loop chunking optimization aims to address this problem.

Motivation. The execution of a parallel loop by work-stealing with the work-first policy proceeds as follows. A worker starts executing the loop by pushing the continuation following the first iteration of the loop onto the deque, making it available for stealing. The worker then executes the first iteration of the loop. Upon completion of the first iteration it checks if the pushed continuation has been stolen, in which case it attempts stealing for more work. Now consider a scenario with two workers, W_1 and W_2. Suppose worker W_1 starts executing the loop by pushing the continuation following the first iteration onto the deque. Worker W_2 steals the continuation and starts executing the second iteration (after pushing the continuation following the second iteration onto its deque). After completing the first iteration of the loop, W_1 finds that its continuation has been stolen and looks to steal work from W_2. In this case, W_1 will steal the continuation from W_2 and start executing the third iteration, and so on. The continuation will this alternate between the two workers and the number of steals could be as high as the number of iterations. Dynamic loop chunking aims to reduce the number of steals in such scenarios, thereby reducing the performance overhead.

Optimization. When a worker pushes a continuation following the current iteration into the deque, it effectively cedes the remaining iterations to the thief. In our approach, instead of giving up all the remaining iterations, the current worker gives up only half the remaining iterations to the thief, keeping the other half for itself. Hence, when the current worker's continuation gets stolen, the worker does not have to resort to stealing, as it already has a part of the remaining loop iterations available on the local deque. This way both workers have about the same amount of work to do without needing to steal. Note that after the steal the iteration space of both the workers can be later split again for stealing.

```
A() {
  startFinish();
  for (int i = 0; i <= N; i++) {
    frame.i = i;
    pushFrame (frame);
    ... // execute async
    if (popFrame())
      return
  }
  stopFinish();
}
```

(a) Without Dynamic Loop Chunking

```
A() {
  startFinish();
  for (int i = 0; i <= N; i++) {
    frame.i = (i+N)/2;
    pushFrame (frame);
    ... // execute async
    if (popFrame()) {
      N = (i+N)/2;
      frame = new Frame();
    }
  }
  stopFinish();
}
```

(b) With Dynamic Loop Chunking

Fig. 5. The optimized and unoptimized versions of a *foreach* loop in HJ compiled for work-first work-stealing

[1] While it is possible to use a divide-and-conquer implementation for parallel loops, there are limitations in doing so, depending (in part) on the number of iterations and number of processors.

Figure 5 shows the optimized and unoptimized version of a *foreach* loop in HJ that is compiled for work-stealing with the work-first policy. Both versions show only those parts of the translated code are relevant to this optimization. The unoptimized version executes every iteration by storing the current value of the index variable i into the frame and pushing the frame onto the deque. When this pushed continuation is stolen, the thief starts executing from the next iteration. When the victim finds that its continuation has been stolen, it exits from this method, and starts stealing work. The optimized version, instead of storing the current value of the index variable i, stores the mid point of the remaining iteration space of i into the frame. Now when the continuation is stolen, the thief gets to execute the iterations starting from this mid point, until the current iteration space's end point. When the victim finds its continuation stolen, it does not exit from the method, but instead executes the iterations until the mid point of the current iteration space. This way the thief executes the second half of the remaining iterations while the victim executes the first.

Applicability. This optimization is performed only on those HJ *foreach* loops that operate on rectangular regions, thereby making it possible to convert them into normalized counted loops. Also, the *foreach* loop must have a *finish* construct immediately enclosing it, ensuring that even though both the thief and the victim execute different parts of the iteration space in parallel, only one of them can proceed with the code following the loop. The worker that reaches the *stopFinish* first creates a continuation and exits the method. The worker that reaches the *stopFinish* last proceeds to execute the continuation. Also, we do not discuss the impact of exception semantics on chunking in this paper, since our approach is to adapt the approach proposed in [22] for static loop chunking to our dynamic loop chunking scenario.

Another issue concerns the splitting of the iteration space when the number of iterations is very small. In the example shown in Figure 5, the iteration space is split evenly even when the number of remaining iterations is zero. This still results in correct code, since we just create additional tasks with zero-iteration loops. This can be avoided by introducing a threshold for splitting iterations, which would be checked at runtime. The downside of this check is that since it would have to be performed in every iteration, adding significant overhead if there is a large number of short iterations. Intuitively, the first approach (creating zero-iteration tasks in the case of a very small iteration space, but not introducing any runtime checks) seems to be better since loops with a large iteration space are more likely to have a bigger impact on overall application performance and should be given priority in optimization.

One drawback of this optimization is that we create new activation frames every time a steal occurs. This is necessary because the optimized version will have multiple workers working on different iterations in parallel and each such worker would need an activation frame for the enclosing function. The overhead due to the creation of new frames is relatively small when the number of iterations in the parallel loop is high, but could be significant for loops with a small number of iterations. Again, we believe that loops with large iteration space should be given a priority in optimization.

While we have explained the dynamic loop chunking optimization using an example with only two workers, the optimization generalizes well for any number of workers. The key is to push $\frac{1}{p}$ of the remaining iterations, where p is the number of workers

working on the loop. This way, the optimization dynamically adapts to the number of workers available and the number of iterations left to be executed in the parallel loop.

4.2 Redundant Frame-Store Optimizations

```
    Foo(int x, int y) {
S1:    int b;
S2:    final int a = f1(x,y);
S3:    async f2(a);
S4:    b = f3(a,x);
S5:    async f4(a);
S6:    return f5(a,b);
    }
```

(a) Example HJ code snippet

```
    Foo(int x, int y) {
S1:    int b;
S2:    final int a = f1(x,y);
       frame.x = x;
       frame.y = y;
       frame.a = a;
       frame.b = b;
S3:    async f2(a);
S4:    b = f3(a,x);
       frame.x = x;
       frame.y = y;
       frame.a = a;
       frame.b = b;
S5:    async f4(a);
S6:    return f5(a,b);
    }
```

(b) Code transformed for work-first work-stealing

Fig. 6. Example hj code transformed for work-first work-stealing execution

The compilation techniques for work-stealing parallel runtime systems discussed thus far follow a strategy in which the activation record for a function is stored in a heap object, called a *frame*, which in turn can be used by a thief to execute the continuation in that function. This strategy requires that the fields in the *frame* be up to date before every continuation point, so that the thief accesses the most recent values of local variables when it starts executing the continuation. The standard approach [16] dictates that all the local variables be stored in the *frame* before every continuation point. We observe that this is not strictly necessary, since some of the variables would already be in the frame and some may not be live in the continuation. This observation is analogous to identification of redundant spill store instructions in register allocation.

Figure 6 (a) shows an example HJ code snippet which will be used to illustrate the analyses needed to identify redundant stores. The statements *S3* and *S5* are asyncs and hence there are continuation points starting at statements *S4* and *S6*. Also, let us assume that the functions *f1*, *f3*, and *f5* are not potentially parallel functions so there are no continuation points after calls to these functions.

Figure 6 shows an example of frame-store statement insertion for scheduling with work-first policy. Since there are continuation points after statements *S3* and *S5*, all the local variables have to be stored into the frame before these statements.

Live Variables Analysis. The local variables are stored in the frame before every continuation point so that the thief executing the continuation can use their values. If a local variable is never used beyond a continuation point, the store is not needed. This can be modeled using live analysis, i.e., only those variables that are live beyond the continuation point need to be stored into the frame before that continuation.

```
    Foo(int x, int y)
S1:    int b;
S2:    final int a = f1(x,y);
       frame.x = x;
       frame.y = y;
       frame.a = a;
       frame.b = b;
S3:    async f2(a);
S4:    b = f3(a,x);
       frame.x = x;
       frame.y = y;
       frame.a = a;
       frame.b = b;
S5:    async f4(a);
S6:    return f5(a,b);
```

(a) Marked by Liveness Analysis

```
    Foo(int x, int y)
S1:    int b;
S2:    final int a = f1(x,y);
       frame.x = x;
       frame.a = a;
S3:    async f2(a);
S4:    b = f3(a,x);
       frame.a = a;
       frame.b = b;
S5:    async f4(a);
S6:    return f5(a,b);
```

(b) Marked by Available Expression Analysis

Fig. 7. Redundant Stores

The data-flow equations for the Live Variables analysis are shown in Equations 1 and 2. The set *UE(b)* refers to the set of variables which have an "Upward Exposed Use" in the basic block *b*, *Kill(b)* refers to the set of variables that are "killed" in the basic block *b*, *LiveIn(b)* is the set of all variables that are "live" on entry to the basic block *b*, and *LiveOut(b)* refers to the set of all variables that are "live" on exit from *b*.

$$LiveIn(b) = UE(b) \cup (LiveOut(b) - Kill(b)) \tag{1}$$

$$LiveOut(b) = \bigcup_{s \in succ(b)} (LiveIn(s)) \tag{2}$$

Figure 4.2 (a) shows the frame-stores that are marked for deletion after performing Live Variables analysis on the code. The local variable *y* and *b* are not live in the continuation starting at *S4* and hence they need not be stored into the frame before *S3*. Similarly both *x* and *y* are not live beyond *S5* and hence need not be stored before *S5*.

Available Expressions Analysis. The frame-stores of the local variables that have already been stored into the frame before a continuation point along all paths that reach this frame-store and have not been modified since the last store, are redundant as well. The previous store of the local variable updated the frame with its latest value so the current frame-store can be removed. In order to do this, we need to find out if a local variable is redefined after it was last stored into the frame along any path. A way to model this is to represent the use of a local variable in a frame-store statement as an available expression.

The data-flow equation used to perform the Available Expressions analysis is given in Equation 3. The set *DEExpr(b)* refers to the set of expressions defined in the basic block *b* and not subsequently killed in *b*, *ExprKill(b)* refers to the set of expressions that are killed in the basic block *b*, and *Avail(b)* is the set of all expressions that are "available" on entry to the basic block *b*.

$$Avail(b) = \bigcap_{p \in pred(b)} (DEExpr(p) \cup (Avail(p) - ExprKill(p))) \tag{3}$$

We use the results of the Available Expressions analysis only to check if the trivial expressions involving the local variables are "available" at the frame-store statements. Figure 4.2 (b) shows the frame-stores that are marked for deletion after performing Available Expressions analysis on the code. The frame-store of the local variable *a* before statement *S5* is redundant since there was a store of *a* in to the frame before statement *S3* and *a* has not been modified since the store. The analysis identifies that the expression *a* is available at the frame-store point before *S5*, and marks the frame-store as redundant.

4.3 Objects-As-Frames Optimization

In this section, we introduce a novel optimization that uses application objects to store the additional *TaskFrames* information under the help-first policy for work-stealing, when legal to do so. The idea behind this optimization is to use application objects for storing the *TaskFrame* information, thereby reducing the overhead for separate *TaskFrame* object instantiation, which can be quite costly. The challenge is to find an application object on which to piggy-back the *TaskFrame* information for an async.

Applicability. In our approach, using an application object as a *TaskFrame* involves extending the type of the object to include the fields and methods corresponding to a *TaskFrame*. Since every *TaskFrame* is specialized for a particular async, and we do not allow a *TaskFrame* to store information for more than one async, therefore two objects of the same type cannot be used to represent the *TaskFrames* of two different asyncs. Also, since a new instance of a *TaskFrame* has to be created for every execution instance of an async, a single object cannot be used as a *TaskFrame* for more than one execution instance of the async. To enforce this restriction, we add a *used* flag to the object. We then use this flag to dynamically determine if a particular object has already been used as a *TaskFrame*. We would like to point out one limitation of this approach: since there is no multiple inheritance in Java or HJ, only the top-level objects (objects that extend the Object class) can be used as frames with this approach. Fortunately, in practice many objects that get passed to an async fall into this category, so it is not hard to find an object onto which piggy-back the *TaskFrame* information.

Code Transformation. The optimization proceeds as follows. For every async in the application, we inspect the list of local variables that are passed as parameters to the async. If there is a variable *v* of a user-defined type that has not been used as a *TaskFrame* before, then we use *v* to piggy-back the *TaskFrame* information for this async. To use *v* as a *TaskFrame* we extend the type of *v* to include the methods and fields needed by a *TaskFrame*. It is also extended to contain a boolean flag *used* to indicate whether an object has already been used as a *TaskFrame*. The point where the *TaskFrame* is created is now protected by a conditional to check the *used* flag. If the object has already been used, then we create a new *TaskFrame*. Otherwise, we use the object to piggy-back the *TaskFrame* information for this particular instance of async, storing the remaining parameters of the async into the object itself.

Figure 8 shows the optimized version of the fast clone of the Fib example, transformed for work-stealing with the help-first policy. Here the variable *x* which is a *Box-*

```
void fibFast (Worker w, BoxInteger z, int n) {
  if (n < 2) {
    z.val = n;
    return;
  }
  fibFrame frame = new fibFrame();
  final BoxInteger x = new BoxInteger();
  final BoxInteger y = new BoxInteger();
  w.startFinish();
  TaskFrame tFrame1;
  if (x.used is true) {
    tFrame1 = new Async1TaskFrame(x, n-1);
  } else {
    x.used = true;
    x.n = n;
    tFrame1 = x;
  }
  w.pushTaskFrame(tFrame1);
  ...
}
```

Fig. 8. Fast clone for the fib function for help-first policy work-stealing optimized using application object as a frame

Integer is being used as a *TaskFrame* for the first async. The modified version of *BoxInteger* now contains a field for the variable *n*, a boolean flag *used*, and the body of the async outlined in the *execute* method. The *TaskFrame* creation for the first async in the fast clone is now replaced with a conditional statement checking the *used* flag of *x*. A new *TaskFrame* is created under the *true* branch. *x* is used as the *TaskFrame* under the *false* case, where the local variable *n* is stored in to *x* and the *used* flag of *x* is set to *true*.

Note that the flag *used* is not actually needed in the particular example in Figure 8 because there is a new instance of the variable *x* being created for every execution instance of the first async in the method. However, it may not be possible to ascertain this fact in general for all the variables.

During the optimization, we extend the type of the object being promoted as a *TaskFrame* to contain a field for every local variable used in the async. There could be other objects of this modified type (or of any of its subtypes in the class hierarchy) in the program which are not being used as *TaskFrames*. These objects still carry the space overhead of the extra fields that are added to promote the type to a *TaskFrame*. Also, the optimization adds a conditional statement to test if an object has already been used as a *TaskFrame*. A topic for future work is to avoid these overheads if we can determine at compile time whether an object can be used as a *TaskFrame* for an async. We are currently also investigating a different approach, which uses leaf objects (objects of the type that is not extended by any other type in the program) to piggy-back the *TaskFrame* information. This approach requires analysis similar to the analysis in Object Inlining optimization [7] and while it complicates the access to the *TaskFrame* data, it does not affect the class hierarchy in a way that the first approach does.

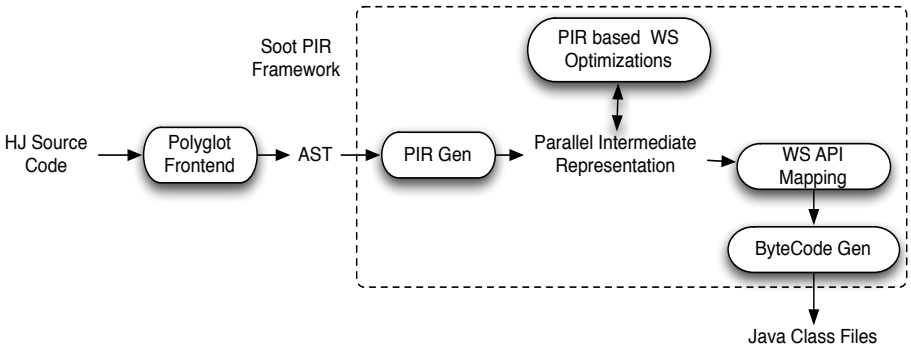


Fig. 9.

Fig. 10. The structure of the Habanero Java compiler

5 Results

5.1 Experimental Setup

All of the work-stealing code generation and optimizations presented in this paper are implemented within the Habanero-Java compiler (HJ) framework. HJ compiler is built upon a Polyglot based front-end and a Soot [24] based back-end (shown in Figure 9). In the back-end, we introduce a new Parallel Intermediate Representation (PIR) extension to the Jimple intermediate representation in SOOT. The PIR includes explicit constructs for parallel operations such as $foreach$, $async$ and $finish$, which simplify compiler transformations involving code manipulation related to the parallel constructs. The work-stealing related translations are performed in WS Optimizations and WS API Mapping, before which the PIR is translated to Java bytecode.

5.2 Performance Analysis of Dynamic Loop Chunking Optimization

This section analyzes the performance of the Dynamic Loop Chunking optimization on the JGF benchmarks. All the results were obtained on a system that includes four Quad-Core Intel E7330 processors running at 2.40GHz with 32GB main memory. The jvm parameters used include: -Xmx2g, -Xms2g, -server.

The aim of the dynamic loop chunking optimization is to reduce the number of steals thereby reducing the overhead. Figure 11 shows the percentage of the number of steals that remain after this optimization, compared to the number of steals without the optimization. As is evident from the graph, this optimization reduces the number of steals to less than 1% in 3 of the benchmarks and to less than 10% in 2 other benchmarks. The reduction in the number of steals is not as high in 'SOR' because the parallel loop in 'SOR' is at the inner level and is enclosed by a sequential outer loop. This optimization reduces the number of steals in the inner parallel loop only. The sequential outer loop adds a constant factor that is not affected by the optimization. Though benchmarks like 'LUFact' and 'Moldyn' also have a similar structure, the constant factor in the outer sequential loop in these cases are not as high as in 'SOR'. The reason for a

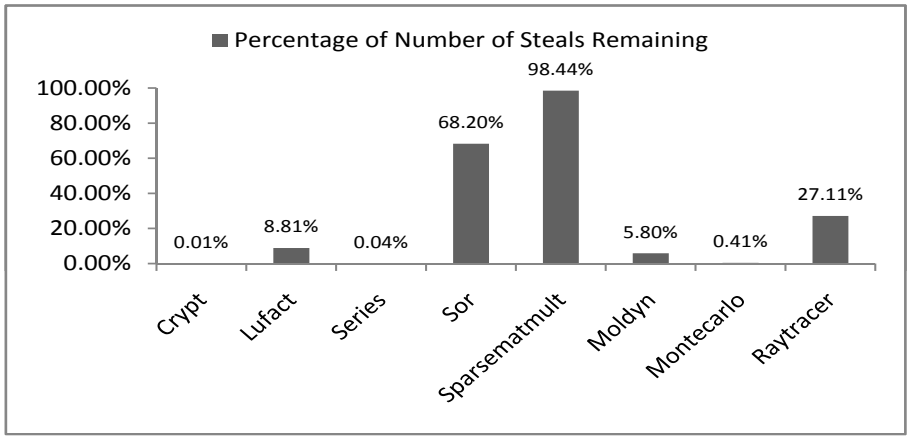


Fig. 11. Number of steals remaining after Dynamic Loop Chunking Optimization

negligible reduction in the number of steals in 'Sparsematmult' is due to the fact that the total number of iterations in the outer parallel loop is very small. Since the number of threads is on the order of the number of iterations, there is effectively no reduction in the number of steals.

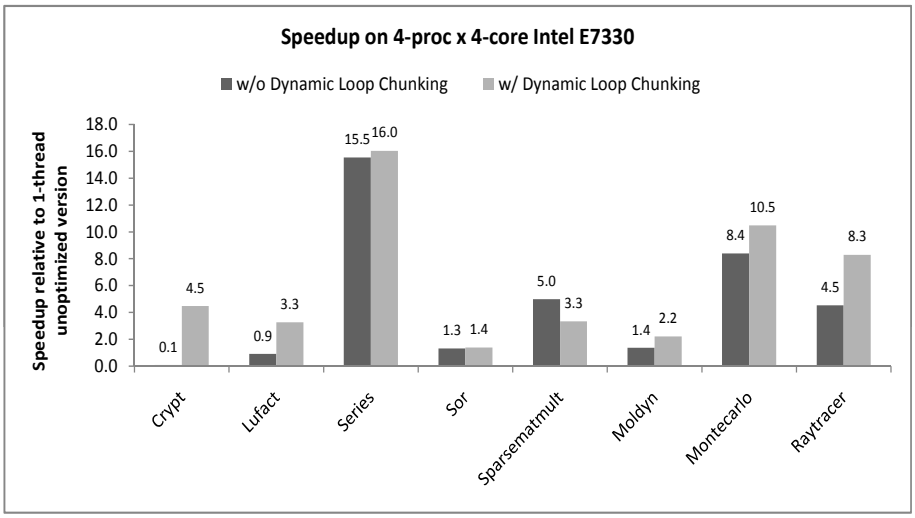


Fig. 12. Speedup with Dynamic Loop Chunking Optimization

Figure 12 shows the speedup of the optimized and unoptimized versions of the JGF benchmarks relative to the 1-thread unoptimized version on a 16-way Intel processor. We observe an improvement in all the benchmarks except 'Sparsematmult'. In benchmarks 'Crypt' and 'LUFact' the performance, which was worse for 16-thread than it

was for a single thread, improves considerably after our optimization. The negligible improvement in 'SOR' is due to the same reasons as given above. There is a degradation in the performance of 'Sparsematmult' after optimization, since the optimized version just adds to the overhead by creating additional frames without reducing the number of steals. This can be avoided by not performing dynamic loop chunking when the number of iterations in the parallel loop is on the order of the number of worker threads.

5.3 Code Size Reduction due to Redundant Frame-Store Optimization

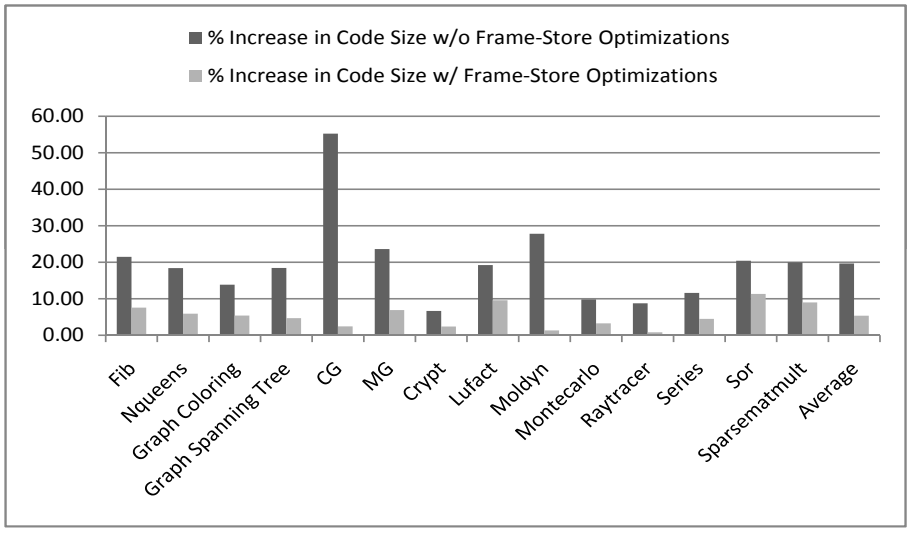


Fig. 13. Increase in Code Size due to work-stealing compilation (in terms of the number of Soot's *Jimple* instructions), with and without Frame-Store Optimizations

We now evaluate the performance of the Frame-Store optimizations on the code transformed for work-stealing schedulers on a wide variety of benchmarks including seven Java Grande Forum (JGF) benchmarks, two NAS Parallel Benchmarks (NPB), and the Fibonacci, N-Queens, Graph-Coloring, and Graph Spanning Tree micro-benchmarks. The JGF and NPB benchmarks are more representative of iterative parallel algorithms. Fibonacci and N-Queens micro-benchmarks have been used in the past [16] to evaluate the performance of work-stealing schedulers. We include the graph algorithms in our evaluations since there has been some attention to scheduling such applications using work-stealing [11].

We studied the reduction in code size (in terms of the number of *Jimple* instructions) due to the frame-store optimizations. In order to analyze the effect of these optimizations on code size, we statically count the number of frame-store instructions that were added during compilation for work-stealing with the work-first policy and the total number of *Jimple* instructions in every potentially parallel function. We also count the number of frame-store instructions that remained in these functions after Frame-Store

optimizations. Figure 13 gives the percentage increase in code size of the potentially parallel functions in the benchmarks with and without Frame-Store optimizations. We observe a reduction in code size as high as 52% for the potentially parallel functions of the 'CG' benchmark. On average, there is a 14% reduction in size of the potentially parallel functions across all the benchmarks. The results would be even more dramatic if the baseline was defined for a compiler that performs no interprocedural analysis and assumes that each function is potentially parallel.

5.4 Performance Analysis of Objects as Frames Optimization

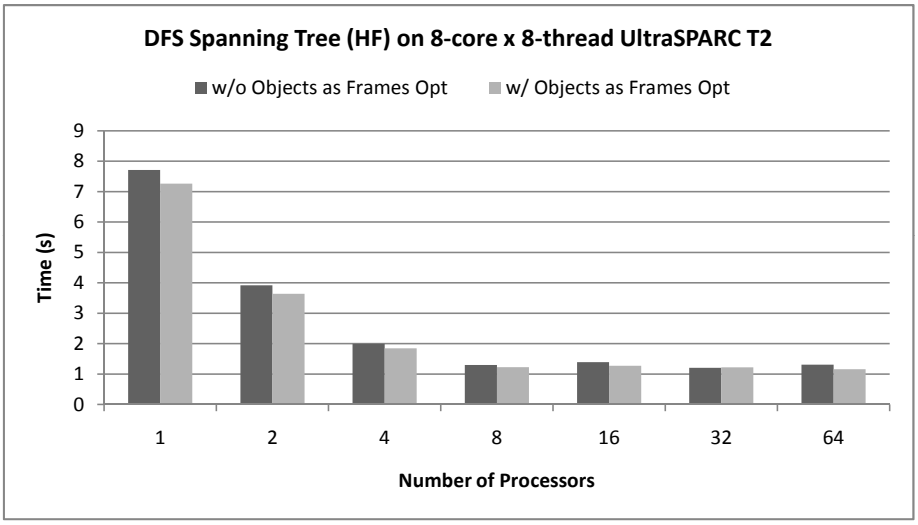


Fig. 14. Objects-As-Frames Optimization on Graph Spanning Tree

We use the Graph Spanning Tree benchmark to evaluate the performance of the Objects-As-Frames Optimization. This benchmark constructs a spanning tree of a randomly generated *Taurus* graph by performing a *Depth-First Search* on the graph [11]. The code for this benchmark is given in Figure 1, showing an async for every neighboring Vertex that has not been traversed already. This essentially means that there is one async for every vertex in the graph. This suggests that the vertex object can be used as the *TaskFrame* for these asyncs. Our optimization does exactly that.

Figure 14 shows the execution time of the optimized and the unoptimized versions of the DFS Graph Spanning Tree benchmark on 64-way UltraSPARC T2. The execution times reported are the minimum among 5 runs for each of the cases. We see an improvement of 5-8% on the execution times for 1 to 8 processors. From 8 processors onwards, there is no significant difference in the execution times. The reason is that the overhead of the creation of new *TaskFrames* that were avoided by this optimization is linear in the number of vertices in the graph, as seen in the 1-processor case. When the number of processors increase, this overhead is distributed among all the processors

involved and becomes relatively smaller. Also, as described earlier, the overheads due to this optimization reduce the gains as the number of processors increase.

6 Related Work

Work-stealing schedulers have a long history that includes *lazy task creation* [18]. Blumofe et al. defined the fully-strict computation model and proposed a randomized work-stealing scheduler with provable time and space bounds [4]. An implementation of this algorithm with compiler support for Cilk was presented in [16]. Agarwal et al. proved that terminally-strict parallel programs can be scheduled with a work-first policy and achieve the same time and space bounds as fully-strict programs [1]. To the best of our knowledge, our work presented in [17] is the first work stealing implementation including compiler and runtime support for an async-finish parallel language which allows escaping asyncs and sequential calls to a parallel function[2].

The X10 Work Stealing framework (XWS) is a recently released library [11] that supports work stealing for a subset of X10 programs in which sequential and async calls to the same function and nesting of finish constructs are not permitted. The library interface requires the user to provide code for saving and restoring local variables in the absence of compiler support. In contrast, our approach provides a language interface with compiler support for general nested finish-async parallelism, and our runtime system supports both work-first and help-first policies.

7 Conclusions and Future Work

Though work-stealing has received a lot of attention in past work, it has primarily been focused on spawn-sync computations. Recently, we introduced extensions to work-stealing runtime techniques to support async-finish parallelism as well as both work-first and help-first policies [17]. These runtime extensions require significant compiler support.

In this paper, we described the compilation techniques needed to support these work-stealing extensions, and several optimizations to further improve the performance of the code generated for work-stealing. In order to support sequential calls to parallel functions, we extended the heap *frame* data structure to maintain the complete call stack. We also perform an inter-procedural analysis by building the call-graph to identify potentially parallel functions. We support arbitrarily nested asyncs by introducing a new *frame* for every task containing a continuation. A topic for future work is to extend work-stealing to support other parallel constructs in HJ, such as phasers [23].

We described the dynamic loop chunking optimization for parallel loops, which aims to reduce the number of steals in the case of work-stealing with the work-first policy. The results show that the number of steals were reduced to less than 1% in three of the JGF benchmarks and to less than 10% in two other benchmarks. This results in

[2] The recent Cilk++ release from Cilk Arts [10] allows the same function to be spawned and called sequentially from different contexts but does not support async-finish parallelism.

very significantly improved scalability of work-first work-stealing execution of code with parallel loops.

The second optimization we described was that of using the application objects for piggybacking *TaskFrames* information for work-stealing with the help-first policy, thus reducing the cost of instantiating new *TaskFrames* for every async. Our results show that there is a performance improvement of 5-8% for 1-8 threads on a 64-way SMP. Currently, the decision to use the application objects as *TaskFrames* is made at runtime. A topic for future work is to extend this optimization so that the decision and replacement of *TaskFrames* with a application objects is done at compile time.

We also performed an optimization to remove the redundant frame-store statements. This optimization showed on average a 14% reduction in code size (based on the number of *Jimple* instructions) of the potentially parallel functions across all benchmarks, with a maximum reduction of 52% for the potentially parallel functions of 'CG'.

References

1. Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.
2. Eric Allan, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, April 2005.
3. Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra, Igor Peshansky, and Vivek Sarkar. Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP), held in conjunction with PACT 2006, Sep 2006*, 2006.
4. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.
5. Architecture Review Board. *OpenMP Fortran Application Program Interface v 3.0*, 2008.
6. Zoran Budimlić, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*, 2009.
7. Zoran Budimlić and Ken Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.
8. B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.
9. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
10. Cilk Arts. *Cilk++ Programmer's Guide Version 1.0.2*.
11. Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

12. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 Object-Oriented Programming, 9th European Conference*, 1995.
13. Joe Duffy. *Concurrent Programming on Windows*. Addison-Wesley, 2008.
14. Jack Dongarra et al. Parallel linear algebra for scalable multi-core architectures (plasma) project. http://icl.cs.utk.edu/plasma.
15. Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, 1995.
16. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.
17. Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium*, 2009.
18. Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 185–197, New York, NY, USA, 1990. ACM.
19. Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.
20. James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
21. Rice University. *Habanero Multicore Software Research project*.
22. J. Shirako, J. Zhao, V. Nandivada, and V. Sarkar. Chunking Parallel Loops in the Presence of Synchronization. In *ICS '09: International Conference on Supercomputing*, 2009.
23. Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.
24. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.