RICE UNIVERSITY

Compiler Support for Work-Stealing Parallel Runtime Systems

by

Raghavan Raman

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

Approved, Thesis Committee:

Vivek Sarkar, Chair
E.D. Butcher Professor of
Computer Science

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering

William N. Scherer III
Faculty Fellow

Houston, Texas

May 2009

Abstract

Multiple programming models are emerging to address an increased need for dynamic task parallelism in applications for multicore processors and shared-address-space parallel computing. 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 thesis, we focus on the compiler support needed to extend work-stealing for dynamic async-finish task parallelism as in X10 and HabaneroJava (HJ). We also 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.

We then propose and implement two optimizations that can be performed in the compiler to improve the code generated for work-stealing schedulers. Performance results show that the Frame-Store optimizations provide a significant reduction in the code size and the number of frame-store statements executed dynamically, but these reductions do not result in execution time improvements on current multicore systems. We also show that the Objects-As-Frames optimization yields an improvement in performance for small number of threads. Finally, we propose topics for future work which include extending work-stealing for additional language constructs as well as new optimizations.

Acknowledgments

I would like to thank my thesis advisor, Prof. Vivek Sarkar, for his support and guidance throughout this work. It is due to his inspiration and continuous encouragement that I was able to successfully complete this work. His enthusiasm in pursuing problems is infectious and that made me work harder to meet the goals. I would also like to thank my thesis committee members, Prof. Keith Cooper and Bill Scherer, for their insights and comments.

I would like to thank my fellow graduate students Raj Barik and Yi Guo who were with me throughout this work. The discussions we had over all the technical details related to this work have been very useful. Many thanks to my lab mates, Dave Peixotto and Jeff Sandavol, who have been of great help to me from my first year at Rice in both technical and non-technical stuff. I thank Jisheng Zhao for his help and suggestions. Thanks to other members of the Habanero group for all the interesting discussions and insights.

My sincere thanks to Vijay Saraswat and Igor Peshansky of IBM Research for the intellectual discussions related to this work.

Finally, I thank all my family and friends. It is because of the constant support of my loved ones that I was able to complete the work successfully.

Contents

List of Figures

Chapter 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 [13], Fortress [2], X10 [14]) 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 Cilk, OpenMP 3.0, Java Concurrency Utilities, Intel Threading Building Blocks, and Microsoft Task Parallel Library. In addition, dynamic data driven execution has been identified as an important trend for future multicore software, in contrast to past programming models based on the Bulk Synchronous Parallel (BSP) and Single Program Multiple Data (SPMD) paradigms. Scheduling algorithms based on Cilk's work-stealing scheduler are gaining in popularity for dynamic lightweight task parallelism but also have inherent limitations.

Scheduling techniques based on work-stealing build on the idea of decoupling

tasks from threads. This decoupling is achieved by creating a fixed set of threads which in turn pick up tasks to work on. Normally, work-stealing schedulers include a runtime library that is responsible for mapping tasks to threads, migrating tasks from one thread to another, and managing the threads themselves. One approach to scheduling applications using work-stealing is to include the calls to the runtime library explicitly in the application. While this technique often achieves very good performance, it places a lot of onus on the programmer. This requires that the programmer is fully aware of the runtime library and the details of scheduler, which in turn affects the productivity. Hence, work-stealing schedulers generally resort to an alternate approach where the parallelism is expressed as tasks at a higher-level of abstraction using some parallel constructs in a language. This code is then transformed into an equivalent version with appropriate calls to the work-stealing runtime library using a compiler. The compiler needs to do a good job of mapping the tasks in the application to threads appropriately in order to match the performance of a good hand-written application with direct calls to runtime. This thesis addresses the challenges involved in the compiler part of work-stealing schedulers.

This thesis focuses on a core subset of the concurrency constructs of the Habanero-Java (HJ) language, which is an extension of X10 $v1.5$, consisting of the async, atomic, and finish constructs. 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 with both the *work-first* and *help-first* policies. We also study the effects of some optimizations performed in the compiler. Though the code generated by the compiler targets the Habanero Work-Stealing Runtime System [23], the techniques described in this thesis can be used to target any work-stealing runtime system in general. This thesis also includes a brief description of our target work-stealing runtime system for

the sake of completeness. The main contributions of this thesis include:

- We implement the compiler support needed to schedule fully strict computations by work-stealing.
- We extend the compilation techniques to support the scheduling of terminally strict computations using work-stealing.
- We describe the changes needed in the compiler to support work-stealing scheduling under the help-first policy.
- We propose and implement data-flow analyses to remove redundant frame-store statements that are inserted by the compiler transformations.
- We introduce an optimization that replaces the use of *TaskFrames* with application objects, under work-stealing with the help-first policy.

The rest of the thesis is organized as follows. Chapter 2 gives a brief introduction to task scheduling and how it has evolved over the recent past. Chapter 3 briefly describes the work-stealing techniques for fully-strict computations and the extensions needed to support terminally-strict computations. This chapter also introduces work-stealing with the help-first policy, which is an alternate strategy to work-stealing. Chapter 4 describes in detail the compiler support needed to support fully-strict computations and the extensions needed in the compiler for terminally-strict computations. It also discusses the changes needed to target work-stealing with the help-first policy. Chapter 5 introduces a couple of optimizations that can be performed in the work-stealing compiler. It also includes a study of the performance of these optimizations. Chapter 6 discusses related work, and Chapter 7 contains our conclusions and suggestions on topics for future work.

Chapter 2

Background

The advent of multicore processors has led to the emergence of different kinds of programming models to use the parallel processing power available. These programming models can be broadly classified into three categories based on their approach to exploit parallelism, namely, the programming language approach, the directives based approach, and the library approach. Irrespective of their approach, an important aspect of these programming models is their support for executing tasks asynchronously. *Tasks* are defined as independent units of work. Since they are independent, the *tasks* give us the flexibility of executing them in parallel. Though *tasks* are independent, they could involve some explicit synchronizations to coordinate with other *tasks*. The challenge is to schedule these tasks efficiently when multiple processors are available.

Earliest attempts at solving this problem [31] proposed that each task be scheduled in an individual thread with the threads synchronizing among them when the tasks need them to. But this approach had serious issues with respect to resource utilization since an increase in the number of tasks might lead to creation of more threads than the operating system can handle. When there are more threads than available processors, some threads sit idle. Idle threads use up a lot of resources like memory

and may also lead to deadlocks and resource overflows. Also, having many threads competing for processors would increase the overhead associated with executing tasks. Since every task is associated with a thread, execution of a task involves creating a new thread, executing the task on the thread and destroying the thread after the task completes. The overhead of creating and destroying threads becomes significant and sometimes even outweigh the gains of executing tasks in parallel, especially in the case of light-weight tasks.

2.1 Work-Sharing

The JUC Executor framework proposed in [31] provides a means of decoupling tasks from threads. A JUC ThreadPoolExecutor creates a pool of threads that are meant to work on the tasks in a program. It also maintains a work queue where the tasks to be executed reside. The tasks that the program creates go to this work queue and the threads pick up these tasks for execution. The executor framework abstracts out the details of how the threads are managed from the program. The thread pool implementations based on the executor framework are flexible to support different policies of managing threads. For example, a fixed-size thread pool implementation creates threads as tasks are submitted, up to the maximum pool size. It maintains the thread pool size constant by adding new threads when a thread dies.

This approach where the newly created tasks are added to a central queue and threads pick up these tasks for execution, is broadly referred to as *work-sharing*. This term essentially comes from the fact that the threads share their work, in the form of tasks, with other threads executing the program.

X10 and Work-Sharing

X10 is a parallel programming language [14] that extends a sequential subset of Java. It provides parallel programming constructs with the aim of increasing the productivity and performance of parallel programming on a wide range of parallel processors. These include, among others, constructs that create tasks to be executed asynchronously, and those that help tasks synchronize among each other.

An implementation of the X10 runtime system based on work-sharing for symmetric shared-memory multiprocessors (SMPs) is described in [3]. It uses an X10ThreadPoolExecutor which is an extension of JUC ThreadPoolExecutor. The main functionality in this extension is the ability of the thread pool executor to add and delete threads depending on the requirement of the X10 runtime system. Hence, the X10ThreadPoolExecutor also has a central work queue which holds the tasks ready for execution. (Actually, the X10ThreadPoolExecutor consists of one work queue per place. Since this thesis focuses on a single place, we only consider the work queue in a place.) It also starts with a fixed number of threads with each thread executing a task. When a task performs a blocking operation, the thread executing the tasks suspends. At this point, the X10ThreadPoolExecutor adds a new thread to the thread pool to compensate for the idle thread. Thus, the number of threads in the runtime system could increase depending on the application being executed.

The work-sharing approach to schedule tasks, as used in the X10 runtime system [3], suffers from the following limitations.

- The main disadvantage of a work-sharing approach is the central work queue data structure. Since all the threads get their work from this work queue, it becomes a scalability bottleneck when the number of threads increase. Also, since the tasks are typically light-weight, the request for task by threads could

be very frequent thereby increasing the contention on the central work queue.

- The strategy used to handle suspended tasks has an inherent limitation. Since threads executing the tasks suspend when the task suspends, new threads are created to compensate for the idle threads. This could increase the number of threads to a point where it is significantly more than the number of available processors. Now, when the idle threads wake up, there would be more threads running than processors available. This would result in contention among threads for processor cycles.

- There is a significant overhead involved with creating and destroying threads. With every new thread that is created, this overhead increases and could eventually become a significant component of the execution time.

2.2 Lazy Task Creation and Work-Stealing

The inherent limitation of work-sharing due to a single central work queue was addressed by this technique called Lazy Task Creation [29]. As the name suggests, this technique creates tasks lazily, i.e., it creates tasks only when there are resources available for the task to be executed. If there are not enough resources available for a new task to execute, the current task continues execution but saves enough information so that the task could be created in the future if a need arises. Thus, when a processor becomes idle, it gets a new tasks from this saved information. This process of an idle processor getting work from a busy processor is known as *work-stealing*.

Under the work-stealing approach, typically one thread is created for every processor available. Since the threads "steal" work from other threads, there is no need for

a central work queue as well. Also since threads are completely decoupled from tasks in this case, a thread need not suspend when a task suspends. It can just proceed with the execution of other tasks. The suspended tasks will be picked up by other threads when they become ready.

2.2.1 Cilk

The Cilk programming language [21] is an extension of C that provides constructs for parallel programming. Cilk uses work-stealing to schedule the tasks in parallel. Though the idea of work-stealing was understood well before, Cilk was the first parallel programming language that featured a provably efficient work-stealing scheduler [4, 7]. The original Cilk-1 language [6] had a restriction that the parallelism must be expressed by the programmer by writing explicit continuations. The Cilk-5 version removed this restriction by introducing succinct, high-level parallel constructs in the programming language which were then translated to continuations by the compiler.

2.2.2 Threading Building Blocks

Intel Threading Building Blocks (TBBs) is a library that enables support for scalable parallel programming using standard C++ [33]. It provides higher-level abstractions to exploit task-based parallelism without considering the underlying platform details and threading mechanisms. The tasks generated by the higher-level abstractions are then scheduled using work-stealing. Threading Building Blocks is also compatible with other threading packages.

Threading Building Blocks provides support for high-level abstractions using generic programming in C++. It supports two different patterns of split/join tasks. The first is where the programmer constructs an explicit "continuation" task that should be

executed as the next task. The second pattern is through implicit continuations. The programmer creates the tasks normally, and the scheduler creates continuation tasks as and when needed. The former is more difficult to program while the latter may sometimes be less efficient in performance because the parent task blocks until its children complete.

2.2.3 OpenMP 3.0

OpenMP 1.0 [9] that emerged as a parallel programming model for shared-memory parallel processors, was focused on extracting parallelism in loops. It uses a directives based approach, where the programmers annotate their programs in C with *pragma* directives that instruct the compilers about the parallelism to be exploited in the program. OpenMP 3.0 specification [10] includes support to handle unstructured parallelism by introducing a tasking model. It introduces a *task* directive to specify independent units of work. The task model specified by OpenMP 3.0 has been implemented in the IBM XL parallelizing compilers [37]. This implementation includes a runtime library that supports thread management, synchronization, and scheduling. It also includes a compiler that transforms the input code into a multithreaded code with calls to the runtime library. Though this implementation does not schedule tasks by work-stealing, it is extremely likely that a future implementation would include support for work-stealing which will also require a similar compiler support.

In this thesis, we discuss in detail the compilation techniques needed to support the extended version of work-stealing for terminally strict computations.

Chapter 3

Work-Stealing Runtime and Extensions

The problem of scheduling multithreaded computations on multicore parallel systems has become a prime research topic since it has a direct effect not only on the utilization of the parallelism available in the machine, but also on the efficiency of parallelization and the corresponding overhead. The work-stealing scheduling paradigm is an important approach to addressing this problem. In work-stealing, the workers that are idle attempt to 'steal' work (tasks) from other workers that have work. These workers are actually threads, typically one per CPU. This idea dates back to at least 1981, when Burton and Sleep [12] introduced the notion of stealing nodes of a process tree for parallel execution of functional programs. Since then, there has been a lot of work in areas related to work-stealing e.g., [25, 19, 20].

The first provably efficient work-stealing algorithm was introduced by Blumofe et.al. [7]. They describe a randomized work-stealing scheduling algorithm for *fully strict* spawn-sync multithreaded computations which is provably efficient in terms of time, space, and communication. In [23], we extended this work-stealing technique

to a broader class of async-finish multithreaded computations, and also explain the support needed during compilation to enable work-stealing for this extended set of *terminally strict* computations.

Our work targets the X10 programming model, with parallel constructs that can be used to extend any sequential language. The X10 v1.5 language [14] is based on a sequential subset of Java, and the open source X10 v1.5 release includes a work-sharing runtime [3]. The Thread Pool Executor library in the work-sharing scheduler is based on Java 5 Concurrency Utilities [31]. More recently, v1.7 of X10 has moved to a Scala-like syntax for source code and richer type system [24]. The HabaneroJava (HJ) programming language that is being developed in the Habanero Multicore Software Research project at Rice University [34] focuses on addressing the implementation challenges for the Java based X10 v1.5 language on multicore processors, with programming model extensions as needed. The focus of this Master's thesis will be on supporting HJ's core async-finish constructs with work-stealing schedulers.

This chapter describes the work-stealing technique in detail and is organized as follows. Section 3.1 summarizes past work on basic work-stealing scheduling and the work-first policy used to support Cilk-style spawn-sync computations. Section 3.2 explains the language features in HJ that need extensions to the past work. This includes extending the technique to support async-finish computations, which is broader than spawn-sync computations. Section 3.3 introduces the *help-first* policy, an alternate approach to work-stealing which yields better performance than the *work-first* policy in many cases.

3.1 Basic Work-Stealing and Work-First Policy

Blumofe et al. [7] defined the notion of fully strict computations as follows. Each multithreaded computation comprises of a set of tasks and can be viewed as a dag of instructions connected by dependency edges. Each of these tasks represent different execution instances of sequential code. The instructions in a task are connected by *continue* edges which represent the sequential ordering constraints for these instructions. The tasks form a spawn tree with *spawn* edges which go from a specific instruction in the parent task to the first instruction in the child task. There are also *join* edges that connect a task to its parent task to enforce dependences due to the sync operation. Any correct schedule of the multithreaded computation should obey the constraints imposed by all three types of dependency edges - *continue*, *spawn* and *join*. A *strict* computation is one in which all join edges from a task go to one of its ancestor tasks in the spawn tree. A *fully strict* computation is one in which all join edges from a task go to its parent task in the spawn tree. A *terminally strict* computation is one in which join edges from a task emanate only from the last instruction of that task.

The Cilk multithreaded programming language [21] is a parallel extension of C that uses work-stealing to schedule its computations on parallel machines. All computations generated by Cilk-style constructs fall in a restricted set of fully strict computations where a *join* edge must originate from the last instruction of a task. Further, if child task *C2* was spawned after child task *C1* by parent *P*, the destination of *C2*'s join edge cannot precede the destination of *C1*'s join edge in *P*. JCilk [27], which is a Java based multithreaded language, inherits most of its features and strategies from Cilk and hence generates only fully strict computations. Blumofe et al. [7] describe a randomized scheduling algorithm for fully strict multithreaded computations.

The Cilk language includes the spawn construct which is used to generate a child task that can be executed asynchronously in parallel with the code that follows the spawn in the parent task. In Cilk, the spawn keyword must be followed by a procedure call which can be executed in parallel. The sync construct allows tasks to wait and synchronize before they proceed. A sync at any point in a function[1] waits for all the tasks that were spawned by this function to complete. The semantics of spawn and sync are close to that of fork and join respectively. There is an implicit sync at the end of every Cilk procedure, which ensures that all computations generated by Cilk are fully strict. The Cilk language also includes inlets, which are essentially C functions nested within a Cilk function. These inlets can be designated to execute at the end of a spawned function call. The mechanism of returning values from spawned procedures to the parent function is handled using implicit inlets which are automatically generated by the Cilk compiler. There is also an abort construct which, when executed inside an inlet, causes all of the already spawned child tasks of the procedure to terminate.

Figure 3.1 shows an example Cilk code with spawn and sync operations and its corresponding computation dag. The task Γ_0 that executes function $A()$ executes $S0$ and then spawns function $B()$ as a separate task Γ_1. Now Γ_0 and Γ_1 can be executed in parallel. The function $B()$ then executes $S1$ and spawns function $C()$ as a new task Γ_2. After this spawn, there are three tasks that can be executed in parallel. Γ_0 has a sync in 4 and will wait at that point till Γ_1 completes. Also, there is an implicit sync at the end of every Cilk function and hence Γ_1 will wait for its child Γ_2 to complete before it returns. Only after Γ_1 completes will Γ_0 proceed to execute $S5$. The computation dag corresponding to the Cilk code shows how the execution of the tasks proceed by maintaining the dependences through *spawn*, *continue* and *join* edges.

[1]The words, function and method, will be used interchangeably in this thesis

```
   A() {
1    S0;
2    spawn B();
3    S4;
4    sync;
5    S5;
6  }
   B() {
7    S1;
8    spawn C();
9    S3;
10 }
   C() {
11   S2;
12 }
```

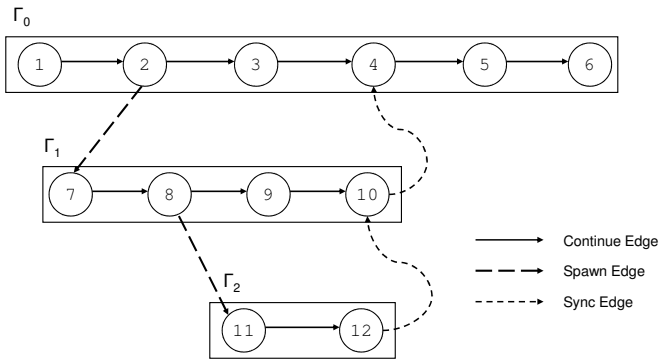


Figure 3.1: Cilk computation dag

The Cilk work-stealing runtime comprises of a set of workers, typically one per CPU or core. Each worker maintains a local *deque* of frames each of which represent work. The runtime starts with one worker executing the main function and the rest being idle with empty deques. Whenever a worker is idle, it becomes a thief and attempts to steal work from another workers' deque. The thieves always steal work from the top of other workers' deques. On the other hand, the workers push and pop their work from the bottom of their own deque.

On a spawn, the worker executes the spawned task. The continuation of the function after the spawn is saved in a frame which is pushed on to the worker's deque so that other workers can steal it. A continuation of a function is simply the same function with a different starting point of execution (and local variables appropriately initialized). Thus, a continuation after a spawn will start from the statement just after the spawned procedure call. Whenever the worker returns from a spawned task, it will first check if the frame that it pushed before the spawn is stolen. If so, the

fully-strict model guarantees that there is no other useful work on the deque for the worker and hence it becomes a thief. Otherwise, it just executes the continuation of the function after the spawn.

The Cilk model of work-stealing states that on any spawn the current worker starts executing the spawned child after storing the continuation following the spawn in a frame and pushing the frame onto the deque. Blumofe et al. provided an upper bound on the space needed by the multithreaded computations that are scheduled according to this technique [8]. Scheduling according to this model maintains the *busy-leaves property*: "at every time step, every living thread that has no living descendants has a processor working on it." The busy-leaves property implies that at all time steps, the portion of the spawn tree that contains only live threads has at most P leaves, where P is the number of threads working on the computation. The space used by each such leaf and all of its ancestors is at most S_1, where S_1 is the stack space needed by the serial execution of the computation. Hence, an upper bound on the space used by any schedule of a multithreaded computation that maintain busy-leaves property is $S_1 * P$.

We call this method of executing the spawned child before the continuation following the spawn as the *work-first* policy since the worker goes on to do its work (on a spawn) as in the case of serial execution. We introduce a dual concept, the *help-first* policy later in Section 3.3.

3.2 Work-Stealing Extensions

The work-stealing technique described above was designed and implemented for the Cilk language which generates fully strict computations. We extend this technique to a broader class of computations that are generated by HJ's finish-async constructs.

This involves handling a variety of issues that arise due to the generality of finish-async constructs. This chapter details these issues and also explains the need for new extensions to enable work-stealing for these constructs.

We start with an overview of the finish and async constructs in HJ, which are derived from X10 [14]. The async statement in HJ creates a new task that can be executed in parallel with its parent. The finish statement is used to enable join points for the descendants. It 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. This is analogous to Cilk's sync construct but there are some interesting differences between the two which are described below. As in the case of Cilk, an HJ computation can also be represented as a dag in which each node corresponds to a dynamic execution instance of an HJ statement, and each edge defines a precedence constraint between the nodes. The first instruction of the main task serves as the root node of the dag. Any instruction which spawns a new task will create a child node in the dag with a spawn edge connecting the async instruction to the first instruction of that child activity. As explained earlier, HJ tasks wait on their descendant tasks by executing a finish statement. We model these dependences by introducing *startFinish* and *stopFinish* nodes in the dag for each dynamic instance of a finish construct and creating *join* edges from the last instruction of the spawned tasks within the scope of finish to the corresponding *stopFinish* instruction in the dag.

Figure 3.2 shows an example HJ code and its corresponding computation dag. The task Γ_0 that executes the function $A()$ enters a new finish scope which is modeled by introducing *startFinish* (node 2 in Figure 3.2) in the dag. It then creates a new task Γ_1 for the async in statement 3 which can then go on to execute in parallel with Γ_0. Γ_1 then goes on to create another task Γ_2 for the async in statement 5. Now all the three tasks can execute in parallel. Though the task Γ_1 can run to completion

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

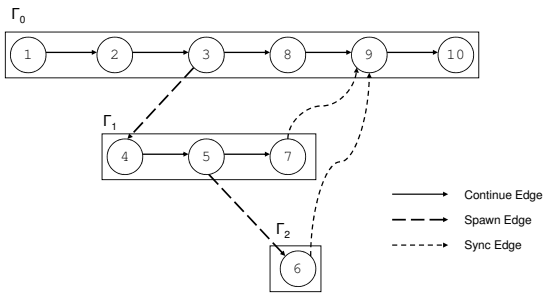


Figure 3.2: HJ computation dag

without waiting for its child task Γ_2 to complete, the task Γ_0 has to wait at the *stopFinish* point (node 9 in Figure 3.2) for all its descendants, in this case Γ_1 and Γ_2, to complete before it can proceed to execute the next statement. This dependency is represented in the computation dag by the *join* edges that go from the last statement in the tasks Γ_1 and Γ_2 to the *stopFinish* node.

3.2.1 Terminally Strict Computations

In HJ, it is possible for a descendant task to outlive its parent. For instance, in the HJ example in Figure 3.2 the descendant task Γ_2 can continue executing even after its parent Γ_1 completes because Γ_1 does not have an enclosing finish and hence it does not need to wait for its descendants to complete. In fact this holds true for all its ancestors, i.e., Γ_2 can outlive all its ancestors as long as they do not have an enclosing finish around the function call that leads to this task. However, the main function in every HJ program contains an implicit finish which ensures that all the descendant tasks complete before the program terminates. This degree of asynchrony can be useful in parallel divide-and-conquer algorithms so as to permit sub-computations at

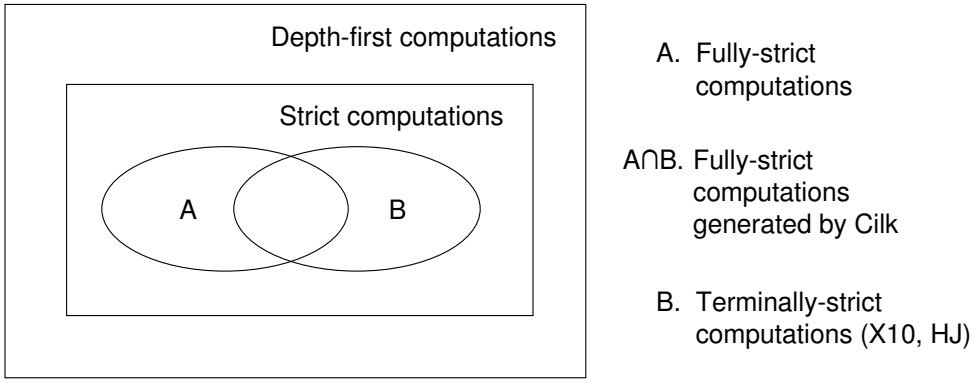


Figure 3.3: Classes of multithreaded computations

different levels of the divide-and-conquer tree to execute in parallel without forcing synchronization at the parent-child level. It can also simplify the creation of parallel iterations over irregular data structures.

Terminally strict computations were introduced in [1] to refer to the class of computations generated by HJ's finish-async style parallelism. The word *terminally* is used to emphasis the fact that the source of the *join* edge can only be the last instruction of a task. This is also true for the class of fully strict computations generated by Cilk. Figure 3.3 shows the relationship between different classes of multithreaded computations. *Depth-first* computations are those for which a left-to-right depth-first schedule is sufficient to meet all its dependence requirements [5]. All computations discussed in this thesis belong to this category since the sequential stack-based execution of both HJ and Cilk programs is sufficient to satisfy the async-finish and spawn-sync dependences. The class of strict computations is a subset of the class of depth-first computations. The classes of fully-strict computations and terminally-strict computations are subsets of strict computations. The class of HJ computations corresponds to the set of terminally-strict computations and the class

of Cilk computations is a subset of fully-strict computations. Also the set of HJ computations is a non-trivial superset of the set of Cilk computations.

Blumofe et al. [7] proved that fully-strict computations can be scheduled with provably efficient time and space bound using work-stealing. They state that:

> The expected time to execute a fully strict computation on P processors using the work-stealing scheduler is $T_1/P + O(T_\infty)$, where T_1 is the minimum serial execution time of the multithreaded computation and T_∞ is the minimum execution time with an infinite number of processors. Also, the space required by the execution is at most S_1P, where S_1 is the minimum serial space requirement.

The same theoretical time and space bounds have been extended to terminally-strict computations by Agarwal et al. [1]. Since HJ is based on X10 v1.5, its implementation inherited X10's work-sharing runtime. But, prior to the work reported in this thesis, there has been no work-stealing implementation released with compiler support for X10 and HJ.

3.2.2 Escaping Asyncs

The extension of work-stealing to support the class of terminally strict computations that are generated by HJ's finish-async constructs, demands the need to handle some key features that are a result of this broader class of computations. *Escaping asyncs*, which refer to tasks that outlive their parents, are one important aspect of HJ that is different from Cilk and demands additional compiler and runtime support for work-stealing.

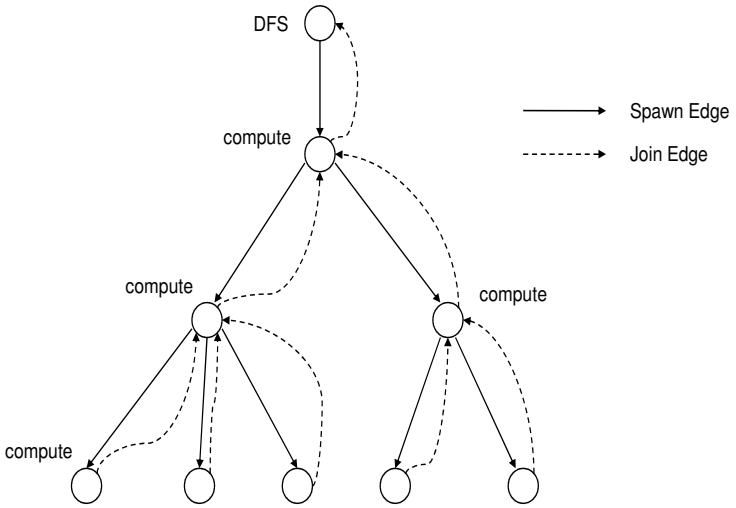
Let us consider the parallel DFS spanning tree graph algorithm [16] whose HJ code is shown in Figure 3.4. In a terminally strict language like HJ, a single finish scope

```
 1 class V  {
 2   V [] neighbors;
 3   V parent;
 4   V (int i) {super(i); }
 5   boolean tryLabeling(V n) {
 6       atomic if (parent == null)
 7             parent = n;
 8       return parent == n;
 9   }
10   void compute() {
11     for (int i=0; i<neighbors.length; i++) {
12       V e = neighbors[i];
13       if (e.tryLabeling(this))
14         async e.compute(); //escaping async
15     }
16   }
17   void DFS() {
18     parent = this;
19     finish compute();
20   }}
```

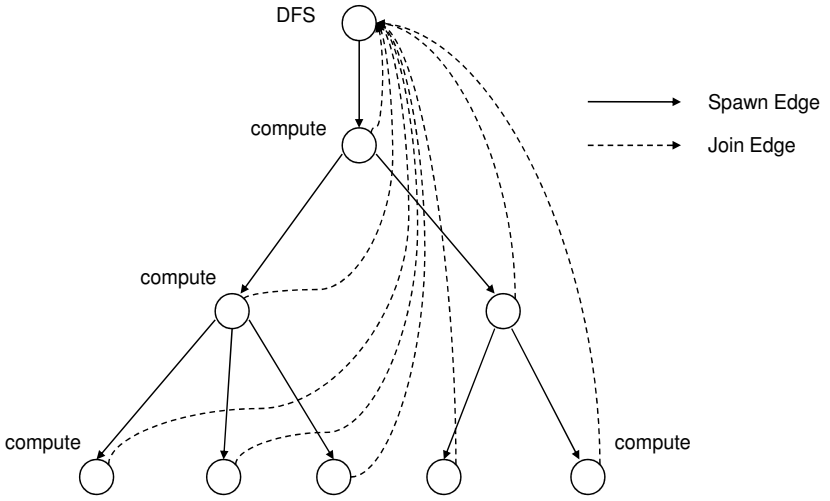
Figure 3.4: Code for parallel DFS spanning tree algorithm in HJ

at line 19 suffices for all descendant tasks spawned at line 14. The spawned tasks can still be alive after its parent (*compute*) returns. However, a fully strict computation implementation, like Cilk, will automatically insert an implicit sync operation at the end of each function thereby ensuring that every parent waits for all its children to complete. Semantically, it is equivalent to adding a finish scope to enclose the body of the *compute* function.

Figure 3.5 shows the spawn tree of Cilk's fully strict computation and HJ's terminally strict computation for the program in Figure 3.4. The solid lines are the *spawn* edges and the dotted lines are *join* edges. Note that in a terminally strict computation, all the join edges go from the task to an ancestor in the tree, whereas in a fully strict computation all the join edges go from the task to its parent.



(a) Cilk



(b) HJ

Figure 3.5: Spawn tree for DFS spanning tree algorithm for Cilk and HJ

Space advantage in terminally strict computations

As we have already seen, in a terminally strict computation, a task may terminate without waiting for its descendants to complete. In a managed runtime environment such as Java (which is HJ's runtime environment) this allows the garbage collector to collect the space used by terminated parent tasks even while its children are executing. Whereas in the case of fully strict computations, the garbage collector will not be able to collect the space used by the parent tasks because they can never complete until all their children complete. In practice, this may cause the runtime to exhaust its space if the depth of the spawn tree exceeds a certain threshold.

The challenge imposed by escaping asyncs in performing work-stealing scheduling is that the task that contains a finish scope has to wait for all its descendants to complete before it executes the next statement after the finish scope. In contrast, other tasks (those that do not contain a finish scope) need not wait for their children to complete. Hence every task that contains a finish scope has to keep track of all tasks that are created in that scope which include not only the tasks that it creates but also other tasks that its children (or descendants) create.

3.2.3 Sequential Calls to Parallel Functions

Parallel functions are defined as those that may spawn tasks either directly or indirectly by calling other functions. In Cilk, sequential calls to parallel functions (known as *cilk functions*) are not allowed. As a result, all functions that may appear on a call path to a spawn operation must be designated as cilk functions and also must be spawned. This restriction has a significant software engineering impact because it increases the effort involved in converting sequential code to parallel code, and prohibits the insertion of sequential code wrappers for parallel code. In contrast,

```
A() {
  B();
L1: ...
}

B() {
  C();
L2: ...
}

C() {
  async D();
C1: ...
}

D() {
  E();
L3: ...
}

E() {
  if (...)
    async E();
C2: ...
}
```

Figure 3.6: Example HJ code: Sequential call to Parallel functions

HJ permits the same function to be invoked sequentially or via an async at different program points.

Figure 3.6 contains a sample HJ code. Note that though the functions $B()$, $C()$, and $E()$ are parallel functions (i.e., they lead to the creation of asyncs), they are called sequentially. This code cannot be directly translated to Cilk. In Cilk, $C()$ and $E()$ would be *cilk functions* because they may spawn tasks and hence they cannot be called sequentially in functions $B()$ and $D()$ respectively. Thus we may have sequential calls to parallel functions in HJ but not in Cilk.

This imposes a challenge in work-stealing scheduling as it introduces new continuation points located immediately after sequential calls to parallel functions. This is because it is no longer necessary for the worker that executed the sequential call to return back to the caller. It may be the case that the parallel function that was called sequentially was stolen by some other worker and hence that worker which executes the last continuation in the callee is the one that will return to the caller. For example, consider the sequential call to the parallel function $C()$ in the HJ code in Figure 3.6. If the continuation $C1$ in $C()$ gets stolen, then the thief will get to

complete $C()$ and hence it is this thief that will return to $B()$, not the worker that executed the sequential call. And this thief will need the stack frame of the original caller to continue execution from the continuation point. Hence there is a need to store the stack frame of $B()$ in a heap object in this case even though $B()$ has no parallel code. The exact details of how this is accomplished are described later in Section 4.2.

3.2.4 Arbitrarily Nested Asyncs

The syntax of spawn in Cilk requires that the new task that is spawned should always be a procedure call. It cannot be used to spawn an arbitrary piece of code and hence it 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 can be used to create a parallel task for any arbitrary piece of code. The only restriction in HJ is that the body of async can only access final local variables in outer scopes. This restriction ensures that an update to an outer local variable in one task cannot be silently communicated to another task. (However, it is still possible for a task to update a field of a shared object and for another task to see the update.)

Figure 3.2 shows an HJ code with nested asyncs. In order to achieve a similar parallelism in Cilk the parallel tasks must be wrapped in a procedure as shown in Figure 3.1.

This arbitrary nesting of asyncs in HJ presents a challenge in scheduling by work-stealing because with this feature there may be more than one worker executing different parts of the same execution instance of a function at the same time. Also, since the nested tasks can start their execution at any point in time (satisfying the synchronization constraints imposed by the enclosing finish) the data for these tasks

need to be stored in a manner that ensures that they are not overwritten by a nested or parallel task in the same function. This is unlike Cilk, where there can be only one worker executing an instance of a function at any point in time.

3.2.5 Delayed Asyncs

The concept of *delayed* asyncs was introduced in HJ by Budimlic et.al as an optimized way to implement CnC's prescription construct [11]. This refers to a parallel task that can only start executing when a condition becomes true. The delayed async statement in HJ, *async when(<cond>) <stmt>*, is similar to a normal async except that the execution of *<stmt>* is guaranteed to be delayed until after the boolean condition, *<cond>*, evaluates to true. The boolean condition guarding the async must be monotonic in the sense that it remains *false* until a point in time after which it changes to *true* permanently. Thus it gives a scheduler a definite point in time after which the async can be scheduled. Also, the boolean condition must be side-effect free since it may be evaluated multiple times before the async is scheduled.

The advantage of delayed asyncs is that they provide an easy way to schedule a task to run in parallel at a later point in time when a condition, such as a data dependency, is satisfied. When scheduled by a work-stealing runtime system, a delayed async requires all the support of a normal async. In addition, it also needs runtime support to quickly check if the boolean condition that guards the delayed async evaluates to true. In particular, a work-first policy will not suffice for delayed asyncs.

3.2.6 Distributed Parallelism

The X10 programming language [14] has the notion of *places* which is a feature that introduces locality explicitly into the language. This gives programmers control

on where every task must execute and which tasks and objects must be co-located. The exact definition of *places* as given in [14] is as follows:

> A *place* is a collection of *resident* (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant here.

The existence of *places* introduces a new challenge in scheduling by work-stealing since there is a need to map the abstract (user-specified) places to actual processors as specified by a given *deployment*. This mapping from abstract places to processors can vary from a simple one-to-one mapping to something more complicated based on the hardware that is available. Also, the work-stealing runtime now has to account for the cost of stealing across places. This is because stealing a task from a worker that is executing in a different place will likely be more expensive than stealing one from a worker executing in the same place as the former would involve migrating objects. From the compilation point of view, the only change that is needed is the support for mappings from abstract places to actual processors. This was not an issue with the work-stealing scheduler for Cilk since the language views all workers uniformly and does not include any feature to support locality explicitly.

3.3 Help-First Policy

According to Cilk's work-stealing strategy [21], a worker executes a spawned task and leaves the continuation to be stolen by another worker. We call this work-stealing with a *work-first* policy because it proceeds by the original worker doing the work in a normal sequential order. In other words, the worker eagerly executes a task when it is created. We now describe an alternate strategy to steal work, which we

```
                  startFinish();                      startFinish();
                  push continuation after L1;         push task S1 to local deque;
                  S1; //Worker executes S1 eagerly    push task S2 to local deque;
finish {          return if frame stolen              S3;
     async S1;    push continuation after L2;         stopFinish();
L1:  async S2;    S2; //Worker executes S2 eagerly
L2:  S3;          return if frame stolen;
}                 S3;
                  stopFinish();

                       Work-first Policy                   Help-first Policy
```

Figure 3.7: Compilation under Work-First and Help-First policies

```
for (i=1 to SOR_ITERATIONS) {
   finish for (p = 0 to P-1) {
      // parallel for.
      // work partitioned into P chunks
      async {...}
   }
}
```

Figure 3.8: SOR code structure depicting iterative loop parallelism

call work-stealing with a *help-first* policy [23]. The *help-first* policy dictates that a worker executes the continuation and leaves the spawned task to be stolen. We use the name "help-first" for this strategy because it suggests that the worker will ask for help from its peer workers before working on the task itself. Figure 3.7 shows the outline of the result of compilation under work-first and help-first policies. Details of the compilation approach for scheduling under help-first policy are provided later in Section 4.3.

3.3.1 Comparing Help-First and Work-First Policies

The work-first policy is designed for scenarios in which stealing is a rare event. However, the overhead of steals can become a significant bottleneck as the number of workers increases. One example is iterative loop parallelism. Figure 3.8 shows the code structure of a wavefront algorithm implemented in an HJ version of the Java

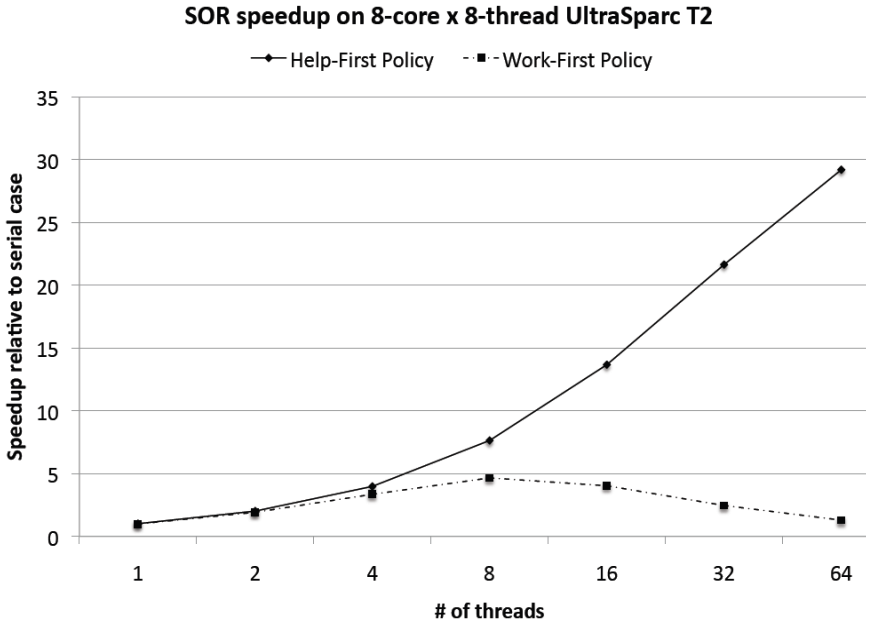


Figure 3.9: Speedup of `SOR` over its serial version on 64-way UltraSparc T2 under work-first and help-first policies

Grande `SOR` benchmark. It has a sequential outer loop and a parallel inner loop. The work in the inner parallel loop is divided evenly among P workers. Figure 3.9 shows the speedup of this `SOR` benchmark relative to its serial version on a 64-thread UltraSparc T2 Niagara2 machine [23]. If the asyncs are executed under the work-first policy, we observe a degradation in performance beyond 8 threads. The following analysis from [23] explains why.

Let the total amount of work in one iteration of the outer loop be T, the time to migrate a task from one worker to another be t_{steal} and the time to save the continuation for each iteration be t_{cont}. Under the work-first policy, one worker will save and push the continuation onto the local deque and another idle worker will steal it. Therefore, distributing the P chunks among P workers will require $P-1$ steals and

these steals must occur sequentially. The length of the critical path of the inner loop is bounded below by $(t_{steal} + t_{cont}) * (P - 1) + T/P$. According to the THE protocol described in [21], a push operation on the deque is lock-free but a thief is required to acquire the lock on the victim's deque before it can steal, thus $t_{cont} << t_{steal}$. As P increases, the actual work for each worker T/P decreases and hence, the total time will be dominated by the time to distribute the task, which is $t_{steal} * (P - 1)$.

When asyncs are scheduled under the help-first policy, the performance of the `SOR` benchmark scales well as shown in Figure 3.9. Under the help-first policy, the worker, upon executing an async, creates and pushes the task onto the deque and proceeds to execute the async's continuation. Once the task is pushed on to the deque, it is available to be stolen by other workers and the stealing can be performed in parallel using non-blocking algorithms. The details on how the stealing happens in parallel at runtime is explained in [23]. Let t_{task} be the time to create and push the task to the deque. For the same reasons as explained above, $t_{task} << t_{steal}$. As the stealing overhead is parallelized, the overhead of the steal operation is not a bottleneck any more and hence the observed improvement in scalability for the help-first policy.

An alternative approach to schedule parallel loops by work-stealing is to use a divide-and-conquer approach, i.e., the thief can be allowed to steal one-half of the remaining iteration space of the loop that it steals work from instead of the current strategy where it steals the entire iteration space that remains. This strategy would help reduce the overhead due to steals for parallel loops. But this would not work for loops whose iteration counts are unknown on loop entry. For example, it would not work for pointer-chasing while loops with `async`s in their body.

Another example where the work-first policy suffers is the parallel Depth First Search (DFS) spanning tree algorithm shown in Figure 3.4. If the asyncs are scheduled under the work-first policy, the worker calls the `compute` function recursively and will

overflow any reasonably sized stack, for large graphs (like its equivalent sequential version). Hence, one cannot use a recursive DFS algorithm if only the work-first policy is supported for asyncs. Since the stack-overflow problem also arises for a sequential DFS implementation, it may be natural to consider a Breadth First Search (BFS) algorithm as an alternative. However, Cong et al. [16] show that, for the same input problem, BFS algorithms are usually less scalable than DFS algorithms due to their additional synchronization requirements. Our approach in such cases is to instead use the help-first policy that has a lower stack size requirement than the work-first policy.

3.3.2 Limitation of Help-First Policy

An important theoretical advantage of the work-first policy is that it guarantees that the space used to run the parallel program is bounded by a constant factor of the space used in its sequential version. In the help-first policy, since we are creating tasks eagerly, the space bound is not guaranteed. There is a possibility to explore a hybrid of these approaches in order to guarantee a space bound while still retaining the benefits of the help-first policy. This can be done by adaptively switching between help-first and work-first policies at runtime depending on the size of the local deques. This is part of the plan for future work in the Habanero project.

Chapter 4

Compiler Support for Work-Stealing

Scheduling HJ's finish-async constructs by work-stealing is a two stage process. The first stage involves the compilation process which transforms the input HJ code into a version that uses the work-stealing runtime. The second stage is the actual execution of the transformed code with a work-stealing runtime that implements the appropriate work-stealing strategy. This chapter explains the first stage, i.e., the compilation process needed to support work-stealing schedulers, which is the focus of this Master's thesis. The work-stealing runtime which this compiler targets, was developed collaboratively with other members of the Habanero project, and is explained in greater detail in [23].

Figure 4.1 shows the high-level compilation structure for HJ programs. The current *front-end* for HJ (which is based on Polyglot [30]) parses the input HJ source code and generates bytecode with all the HJ constructs expanded appropriately to use the HJ `work-sharing` runtime [3]. This step involves the construction of the Polyglot AST for the input source code. The front-end then unparses the AST into

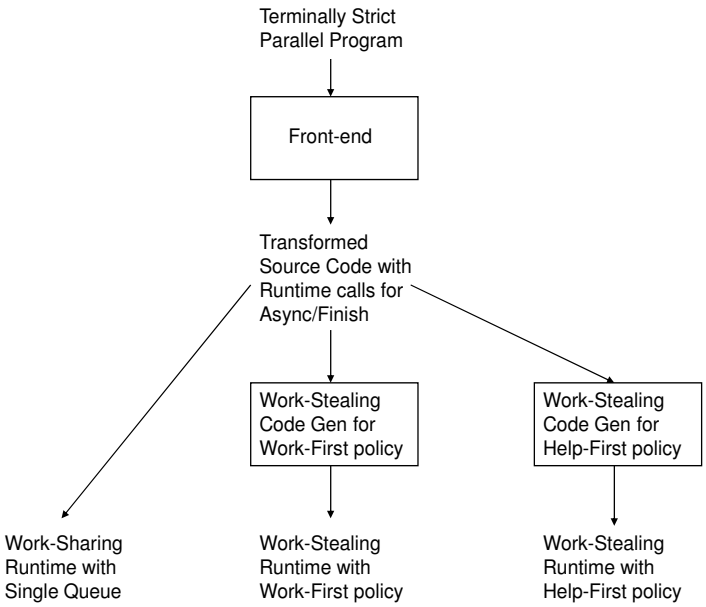


Figure 4.1: Compilation structure for work-sharing and work-stealing runtimes

Java source code with runtime calls inserted to implement the HJ constructs using a work-sharing runtime. For example, an HJ async is expanded to a Java inner class with the body of the async being wrapped in a method of that class. Finally the front-end invokes a Java compiler (*javac*) to translate Java source code to classfiles.

The work-stealing code generation pass then picks up the bytecode generated by the *front-end* and performs the necessary transformations for use in a work-stealing runtime. As shown in Figure 4.1, there are different code generation passes that target the two different runtimes based on work-first and help-first policies. The work-stealing code generation described in this thesis was performed in the *Jimple* intermediate representation of Soot framework [38], which is an optimization framework for Java. The same techniques can be applied to perform these transformations in any other framework that supports the underlying language. Currently, there is

work in progress in the Habanero project to bypass the generation of source code, and instead directly translate the front-end's AST to Jimple [36].

This chapter explains the techniques used in compiler support for work-stealing runtimes for HJ programs. Specifically, we target the work-stealing runtime developed as a part of the Habanero project though the same techniques are applicable to other work-stealing runtime systems. Section 4.1 reviews the basic compilation techniques for work-stealing (by *work-first* policy) of spawn-sync programs as in Cilk and JCilk. Section 4.2 describes our extensions to support additional features in support of terminally strict finish-async constructs. Both these sections discuss compilation techniques needed for work-stealing using the work-first policy. Section 4.3 then details the differences in the transformations needed to target work-stealing using the *help-first* policy. Section 4.4 summarizes the performance results presented in [23] comparing the work-stealing scheduling techniques, the work-first and the help-first policies, with the work-sharing technique.

4.1 Support for Basic Work-Stealing with Work-First Policy

As explained in Section 3.1, basic work-stealing refers to scheduling support for Cilk-style spawn-sync computations. Since the set of terminally strict computations that are generated by HJ's finish-async constructs subsumes the set of fully strict computations that Cilk (or JCilk) generates, the compilation strategy involved in handling the basic constructs remain the same. This section describes these compilation techniques in detail.

4.1.1 The Two-Clone strategy

The transformation of HJ functions to support work-stealing involves generating two clones, a *fast* clone and a *slow* clone for these functions. Every function that contains an async or a finish needs to be transformed. As described later, there could be other functions in HJ that will need these transformations. The details regarding how to select the functions that need to be transformed are given in Section 4.2.

The *work-first* principle given by Frigo et.al [21] states that:

> Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.

where *work* refers to the time taken to execute the computation on one processor and *critical-path* refers to the execution time of the computation on an infinite number of processors. This is accomplished by making one of the clones really fast and the other clone to bear most of the overhead.

Interaction between the two clones

When a worker starts executing a spawned function, it starts with the fast clone of that function. It is the fast clone that will be executed until a part of that function is stolen by another worker. Once a continuation of a function is stolen, the thief starts executing the slow clone of that function. This technique guarantees that the fast clone is never stolen because a worker which steals a function switches to executing its slow clone and will execute the slow clone to completion, i.e., it can never come back to executing the fast clone for that instance of the function. Since a fast clone can never be stolen, it is actually very similar to the sequential version of the code. It just contains some book-keeping code to ensure that the continuations can be stolen. In contrast, the slow clone contains support for migration from one worker to another,

and hence will bear most of the overhead.

Let us consider the restricted case of an async statement wherein the body of the async consists only of a function call like the Cilk spawn statement. In general, an HJ async statement can contain any arbitrary piece of code. The details about how general asyncs are handled are given later in Section 4.2. In a fast clone, each async statement containing a function call body is just replaced by a sequential call to that function. However, the local variables of the current procedure need to be stored in the heap before this sequential call. The finish construct in HJ waits for all its descendants to complete before proceeding on to the next statement. Thus every statement executed in HJ has a unique *Immediately Enclosing Finish* (IEF) dynamic statement instance [35]. Thus the finish construct defines a dynamic scope such that all the asyncs created in this scope report to the IEF. We accomplish this by splitting the finish construct into two operations: one at its start and another at its end. We insert appropriate calls to the runtime at the start and end so that the runtime can keep track of the finish scope accordingly.

Since the process of stealing a task from a worker happens from the top of the *deque* of that worker, it guarantees that the parent tasks are stolen before their child tasks. Thus, whenever the fast clone of a function is being executed, it is guaranteed that the function itself and all its children have never been stolen. Once a function is stolen, the slow clone of that function executes in parallel with its children. In essence, the slow clone is very similar to the fast clone with this additional support for migration across workers, thereby enabling parallelism. In particular it allows the thieves to start executing from any continuation point in that function.

```
 1  void fib (BoxInteger z, int n) {
 2      if (n < 2) {
 3          z.val = n;
 4          return;
 5      }
 6      final BoxInteger x = new BoxInteger();
 7      final BoxInteger y = new BoxInteger();
 8      finish {
 9          async fib(x, n-1);
10          async fib(y, n-2);
11      }
12      z.val = x.val + y.val;
13  }
```

Figure 4.2: Example fib code in HJ

Limitation

Cilk's work-stealing scheduler is designed for the case when there is lots of parallelism and the number of steals is small, so that the fast clone is the one that is expected to execute most of the time. This is true for many divide and conquer parallel algorithms. But, as discussed in Section 3.3 there are other parallel codes for which Cilk's work-stealing scheduler (and HJ's work-stealing with the work-first policy) performs poorly.

4.1.2 Activation Records as Frames in the Heap

The basic strategy behind work-stealing with the work-first policy is that when a worker W_0 is executing a child task, some other worker W_1 can steal the continuation in the parent task and start executing it. The continuation of the parent task is the same function instance with a new starting point for execution, which will be the point after the spawned child call. For the thief W_1 to execute the remainder of the parent, it will need access to the *activation record* of the function instance from the stack of W_0. Since the thief is in a different thread, it cannot directly access

```
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();
    frame.x = x;
    frame.y = y;
    frame.z = z;
    frame.n = n;
    frame.pc = 1;
    fibFast (w, x, n-1);
    if (w.popFrame()) {
        return;
    }

    frame.x = x;
    frame.y = y;
    frame.z = z;
    frame.n = n;
    frame.pc = 2;
    fibFast (w, y, n-2);
    if (w.popFrame()) {
        return;
    }

    w.stopFinishFast();

    z.val = x.val + y.val;
}
```

Figure 4.3: Fast clone for the fib function in Figure 4.2 (unoptimized)

the stack of W_0 without access to the internal stack representation (especially in a strongly typed managed runtime environment like a Java virtual machine). Hence we maintain a copy of the activation frame of the parent (in the same state as when the child was spawned) explicitly in a *frame* data structure in the heap. This *frame* will then be used by W_1 to execute the remainder of the parent task.

The fast and slow clones for the **fib** example in Figures 4.3 and 4.4 show how a *frame* data structure is used to store all the local variables of the function. This

```
void fibSlow (Worker w, Frame f) {
    fibFrame frame = (fibFrame) f;
    int n = frame.n;
    BoxInteger x = frame.x;
    BoxInteger y = frame.y;
    BoxInteger z = frame.z;
    switch (frame.pc) {
    case 0:
        if (n < 2) {
            z.val = n;
            return;
        }
        final BoxInteger x = new BoxInteger();
        final BoxInteger y = new BoxInteger();
        w.startFinish();
        frame.x = x;
        frame.y = y;
        frame.z = z;
        frame.n = n;
        frame.pc = 1;
        fibFast (w, x, n-1);
        if (w.popFrame()) {
            return;
        }
    case 1:
        frame.x = x;
        frame.y = y;
        frame.z = z;
        frame.n = n;
        frame.pc = 2;
        fibFast (w, y, n-2);
        if (w.popFrame()) {
            return;
        }
    case 2:
        frame.x = x;
        frame.y = y;
        frame.z = z;
        frame.n = n;
        frame.pc = 3;
        w.stopFinishSlow();
        if (w.popFrame()) {
            return;
        }
    case 3;
        z.val = x.val + y.val;
    }
}
```

Figure 4.4: Slow clone for the fib function in Figure 4.2 (unoptimized)

frame data structure is unique for every function since it holds the local variables for that function. In this example, the *fibFrame* is the structure corresponding to the function *fib*. In addition to the local variables, the *frame* data structure also contains a field for a pseudo program counter, *pc*. This field is set before spawning a child so that any worker that steals the continuation knows the point in the stolen function at which it should resume execution.

It is worth noting that at any point in time there will be only one worker executing the body of an instance of a function (excluding the body of the asyncs). In other words at any point every function instance can have at most one continuation that is available to be stolen. This invariant ensures that only one instance of the frame is needed for every instance of a function. This approach works for the restricted case when the body of an async is a function call. But asyncs in HJ can contain arbitrary statements in general, in which case a single frame for the entire function is no longer sufficient. The techniques to handle this case are described later in Section 4.2.3.

4.1.3 Continuations as Jumps in the Bytecode

The slow clone should have the property that any worker executing it after a steal must be able to resume execution from any of the continuation points in that method. The continuation point from which the execution must start is dictated by a logical program counter value in the *frame*. In order to achieve this, a *tableswitch* bytecode instruction is inserted at the start of the slow clone, which jumps to the appropriate statement in the code based on the *pc* value in the *frame*. Also, since the code from the continuation point will use the local variables of the method, all the local variables must be initialized with the corresponding values from the *frame* before the *switch* statement.

An important issue to be noted here is that the jump to a continuation point

could target any point in the code (any statement after an async or finish). Hence there is a chance that this jump's target could be a statement within a loop. In this case, the generated jumps will not be valid in Java source code. But since we directly emit bytecode after the transformation, this transformation remains valid as long as it passes the bytecode verifier. Specifically, this transformation has to satisfy the *static constraints* on Java bytecode that relate to jumps. The Java Virtual Machine Specification [28] states that:

- The target of each jump and branch instruction *jsr, jsr_w, goto, goto_w, ifeq, ifne, ifle, iflt, ifge, ifgt, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmple, if_icmplt, if_icmpge, if_icmpgt, if_acmpeq, if_acmpne* must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a wide instruction; a jump or branch target may be the wide instruction itself.
- Each target, including the default, of each *tableswitch* instruction must be the opcode of an instruction within this method. Each tableswitch instruction must have a number of entries in its jump table that is consistent with the value of its low and high jump table operands, and its low value must be less than or equal to its high value. No target of a tableswitch instruction may be the opcode used to specify the operation to be modified by a wide instruction; a tableswitch target may be a wide instruction itself.

The tableswitch instruction inserted by this transformation contains one target for each continuation point in the method, and hence the low and high values of jump table operands in the instruction are consistent with the number of its targets. Also,

the targets of the tableswitch instruction are statements within the same method and hence do not cross method boundaries. Though the transformation happens at the *Jimple* intermediate language level, which is close to Java bytecode, the targets are derived from Java source statements. Hence these targets cannot be the opcode which is an operand of a wide instruction. Thus, this transformation produces bytecode that is consistent with the constraints imposed by the Java Virtual Machine.

4.2 Support for Extended Work-Stealing

This section describes the compilation support needed to extend work-stealing for terminally strict finish-async constructs. In general, this section includes the details regarding how the challenges mentioned in Section 3.2 are addressed. First, we discuss the details regarding potentially parallel functions and techniques to identify such functions. Then, we describe the different approaches to handle sequential calls to potentially parallel functions. Finally, we discuss the transformations needed to handle arbitrarily nested asyncs in HJ. These techniques refer to the compilation support needed for work-stealing with the work-first policy. The modifications needed to support the help-first policy are discussed in Section 4.3.

4.2.1 Potentially Parallel Functions

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 each function body to look for an async or a finish or ensure that the intermediate representation contains summary information indicating if the function contains a finish or an async.

A function is *potentially parallel* if it contains a call to a parallel or potentially parallel function. Thus, any function that contains a continuation is said to be *potentially parallel*. An important point to be noted is that the set of potentially parallel functions is a super-set of the set of parallel functions. It is necessary to identify potentially parallel functions because they have continuation points even though they do not contain any async or finish constructs (as discussed earlier).

Identifying Potentially Parallel Functions

We need to perform a static interprocedural analysis in order to identify the potentially parallel functions. A correct analysis may conservatively err on the side of identifying a function as potentially parallel, even if it is not. Doing so will preserve correctness but could create additional overhead. The first step in the analysis involves building the call-graph of the entire application program. We use the Class Hierarchy Analysis [17, 18] that exists in the Soot framework [38] for building the call-graph. This is a standard analysis that is used to determine, at compile time, a conservative call graph of the entire application. It is implemented by building an internal representation of the hierarchy of classes in the application and using this hierarchy to determine all possible methods that can be the target for a given virtual method call.

Once we have the call-graph, the next step is to traverse it in an efficient manner to identify all the potentially parallel functions. The algorithm to identify the potentially parallel functions involves the following steps. We use the reverse call-graph edges to do the traversal.

- Identify Parallel functions.
- Perform a DFS on the reverse call-graph edges and mark all functions reachable

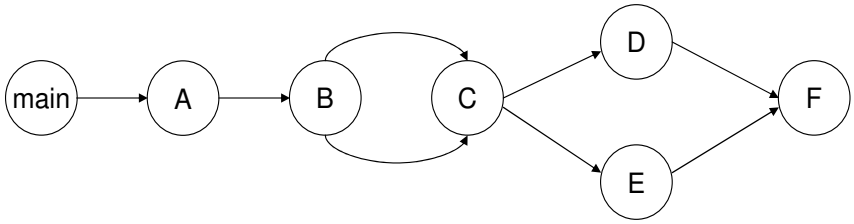


Figure 4.5: An example call-graph

from a parallel function as potentially parallel.

This algorithm identifies the potentially parallel functions in time that is linear in the number of edges in the call-graph of the application. The step to identify the parallel functions is also linear in the number of functions, assuming that each function is annotated with summary information to indicate if it contains a finish or an async. If summary information is unavailable, each test would take time linear in the size of the function to traverse the body of the function in order to check if it contains an async or a finish. Since we perform a DFS traversal of the call-graph to identify potentially parallel functions, it is clearly linear in the number of edges in the call-graph. It also avoids traversing the unnecessary edges, i.e., those edges whose target can never be potentially parallel since we always start from parallel functions.

A possible refinement to this algorithm is to check if a function containing a finish does not indirectly call any asyncs. In this case, the finish can be removed, and no continuation is needed.

For example, consider the call-graph in Figure 4.5. Suppose that the functions represented by nodes D, and E are parallel functions. Note that the algorithm performs a DFS traversal on the reverse call-graph. First it starts from the parallel function D, traverses the reverse edges till the root is reached, marking C, B, A, and

main as potentially parallel. Then it starts from E, but find nothing more to traverse on the reverse edges. The algorithm would not traverse the edges from D to F and from E to F since F can never be potentially parallel.

4.2.2 Sequential Calls to Potentially Parallel Functions

In Cilk, each cilk function corresponds to a task in the computation dag. A parallel function cannot be called sequentially and must be spawned. For the work-stealing scheduler this simplifies the continuation to contain only the activation frame of the current function because the thief that executes the continuation will never return to its caller as its caller must be its parent task whose continuation must have already been stolen. But sequential calls to potentially parallel functions are allowed in HJ. Any such sequential call to a potentially parallel function results in a continuation point immediately after the call for the reasons explained earlier. We now list three approaches to supporting such sequential calls.

Approach 1: Wrap the Call in Finish-Async

A naive way to handle sequential calls to potentially parallel functions is by wrapping the call in a finish-async construct. This would ensure that there is a continuation at the end of the finish, by essentially converting every potentially parallel function into a parallel function. Figure 4.6 shows an example HJ code fragment and the transformed code resulting from this approach. $C()$ and $E()$ are parallel functions while the other functions $A()$, $B()$, and $D()$ are potentially parallel. The sequential calls to $B()$, $C()$, and $E()$ are of interest here. This approach wraps the calls to these functions with finish-async as shown in the figure.

A major 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

Sample Code

```
A() {
  B();
L1: ...
}

B() {
  C();
L2: ...
}

C() {
  async D();
C1: ...
}

D() {
  E();
L3: ...
}

E() {
  if (...)
    async E();
C2: ...
}
```

Transformed Code

```
A() {
  finish async B();
L1: ...
}

B() {
  finish async C();
L2: ...
}

C() {
  async D();
C1: ...
}

D() {
  finish async E();
L3: ...
}

E() {
  if (...)
    async E();
C2: ...
}
```

Figure 4.6: Supporting Sequential calls to Potentially Parallel Functions by wrapping the call in Finish-Async

the sequential call. In the example in Figure 4.6, since the function $E()$ contains an async, the new task that is created and the rest of $E()$ can run in parallel. Similarly the worker that completes $E()$ should be able to continue executing the rest of $D()$ starting at the continuation point $L3$, effectively in parallel with the async in $E()$. By wrapping the call to $E()$ in a finish-async, the worker that completes $E()$ is not allowed to execute the rest of $D()$ and has to suspend.

Another disadvantage of this approach is that it gives rise to extra steals at the continuation points after the sequential calls. A property of the continuations after such sequential calls is that its execution cannot begin until the sequential call completes (though there are no implicit finish statements in that function) because of the semantics of sequential calls. The best way to model this would be to allow the worker that completes the sequential call to execute the continuation in the parent. But according to this technique, since the parent function contains a finish-async around the call, the continuation at the end of the finish is pushed on to this worker's

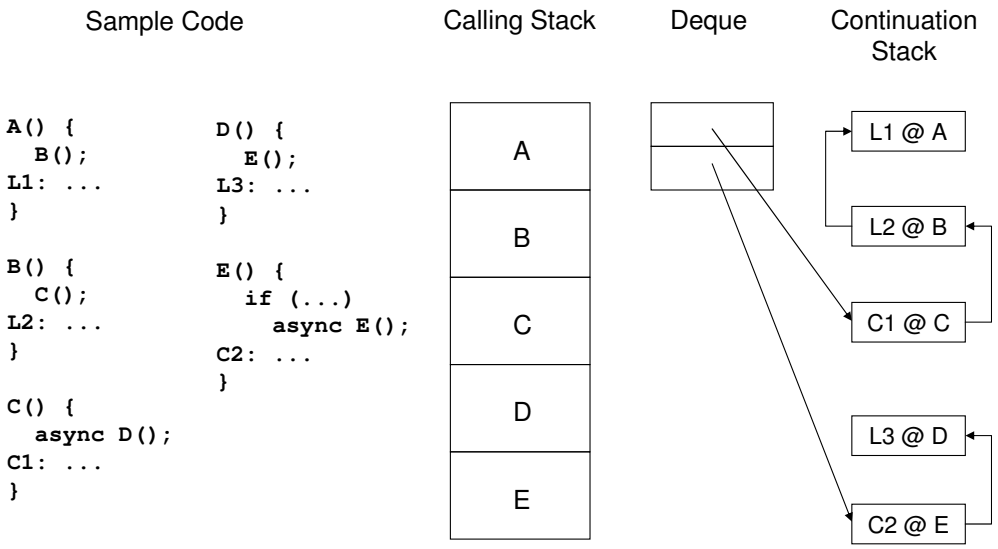


Figure 4.7: Supporting Sequential calls to Potentially Parallel Functions by maintaining the call stack on heap frame

deque first before any other continuation of the callee. Any worker that comes to steal from this worker would end up stealing this continuation first and suspending it immediately since it does not have any work to do. It would then go ahead to steal the next continuation of the child which may have some useful work. Thus every such sequential call could result in an unwanted steal. These overheads can be avoided in the alternative approaches described below.

Approach 2: Maintain the Call Stack on Heap Frame

The sequential calls to potentially parallel functions have a special property that the continuations after such sequential calls can start immediately after the callee returns. There is no other dependency here because it does not involve a finish operation. As mentioned before, one way to model this is to allow the worker that completes the callee to continue executing the continuation in the caller. This way the worker that completes the callee is able to do some more useful work without

stealing. In order to make this possible, every continuation is extended to contain a stack of activation frames up to the previous async in the call chain. There is no need to store any activation frames beyond the previous async because when a worker completes an asynced task it just has to report to the dynamically enclosing finish whose continuation would have been saved already. Thus we do not push the continuations after these sequential calls on to the deque. Instead we attach it to the actual continuation (the ones after an async or finish) which would be executed before this. This approach needs runtime support by which any worker that completes a stolen task looks up the link attached to the task's frame to see if there are more continuations. If there are more continuations, then the same worker proceeds to execute it repeating the same check on completion until there are no more continuations attached.

Consider the example in Figure 4.7. Labels $C1$ and $C2$ 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. This is because it does not have any synchronization points after the sequential call in the caller. Also it avoids a steal at the continuation points after the sequential call since we do not push these continuations on to the deque. We also save the cost of a steal by assigning the worker that completes the callee to execute this continuation in the caller.

The disadvantage with this technique is that it creates activation frames for all

the functions that are potentially parallel and links them together to form a stack of frames according to the call chain. But sometimes these sequential calls may not create continuations, i.e., they may not execute any finish or async. In such cases, their activation frames are redundant.

Approach 3: Copy the stack

One way to avoid the overhead of unnecessary activation frames is to copy the call stack when the stealing occurs [32]. This approach to support sequential calls to potentially parallel functions allows the thief to copy the entire call stack as part of the stealing process. This approach proposes that the execution of asyncs proceed as normal function calls, with some book-keeping operations. The worker thread continues to execute normally. When a thief attempts to steal, it copies the activation stack of the victim upto the first continuation point and start its execution from this continuation point. Thus the thief gets the maximal chunk of work from the victim. But this approach requires access to the runtime stack by user code, which is not allowed by many high-level programming languages, such as Java, for security reasons. So, in order to support this in Java the JVM has to be extended with this functionality. Our approach instead saves local variables in *heap frames* that are manipulated with compiler and runtime support.

4.2.3 Arbitrarily Nested Asyncs

In HJ, an async can contain any arbitrary statement. This is unlike Cilk where the spawn can only contain a function call. Thus in HJ, the parallel task can be a piece of code in the same function as the parent. This allows the user to write code that can have asyncs with arbitrary nesting depths. In order to handle this, the body of the asyncs can be "outlined" into a separate function by the compiler. But the problem

```
   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;
   }
```

Figure 4.8: HJ code to depict the use of *frames* in the case of arbitrarily nested asyncs.

with this approach is that the execution of async will then involve a sequential call and this in turn has to be handled using the techniques described in Section 4.2.2. The other alternative is to retain the body of the asyncs in the parent method and perform the necessary transformations in place. We use this approach in our transformations.

With no constraint on the nesting of asyncs, there can be more than one worker that is executing the same instance of a function, i.e., one worker can be executing the parent task while the other can be executing its child task. Thus having one frame for every function instance will no longer be sufficient. There is a need to have one heap frame per task/activity in order to allow the child tasks in the nested asyncs to run in parallel with their parents. Hence, to translate such functions with arbitrarily nested asyncs into the two clones, we allocate a new frame per task in every function that contains a continuation. This ensures that every new task can execute in parallel with its parent. This also ensures that the same frame is used for all the continuations in a particular task. Figure 4.8 shows an HJ code fragment that contains arbitrarily nested asyncs and an equivalent version with frames created at appropriate points. Note that two frames are being created in this method, one for the method (which is also a task) and the other for the async (which is a child task), because both the

tasks contain continuations.

4.3 Compilation Support for the Help-First Policy

Work-stealing by the help-first policy, explained in Section 3.3, states that the tasks that are created by asyncs are not executed immediately by the worker that creates the tasks. Instead this worker proceeds to execute the continuation after making the new task available to be stolen by other workers. In contrast, in the work-first policy, the worker proceeds to execute the newly created task after making the continuation available to be stolen by other workers. This inherent difference between the work-first and the help-first policies for work-stealing dictates the need for some changes during code generation for help-first work-stealing as compared to that for work-first work-stealing. This section explains in detail all the changes that are necessary to support code generation for help-first work-stealing.

We change the definition of a *parallel function* to reflect the fact that asyncs no longer create continuation points. It is only the end point of finish constructs that result in continuation points for the help-first policy. Though the asyncs need to be handled uniquely in order to make them available for stealing, they will not result in any suspensions or continuation points. Hence, we now define a *parallel function* as one that contains a finish construct. Figure 4.9 shows the fast clone of the *fib* function given in Figure 4.2 when compiled for help-first policy work-stealing. Figure 4.10 shows the corresponding slow clone. The different components of the clones in the example are explained below.

```
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 = new Async1TaskFrame(x, n-1);
    w.pushTaskFrame(tFrame1);

    TaskFrame tFrame2 = new Async2TaskFrame(y, n-2);
    w.pushTaskFrame(tFrame2);

    frame.x = x;
    frame.y = y;
    frame.z = z;
    frame.pc = 1;
    w.pushFrame(frame);
    w.stopFinishFast();
    if (w.popFrame()) {
        return;
    }

    z.val = x.val + y.val;
}
```

Figure 4.9: Fast clone for the fib function for help-first policy work-stealing (unoptimized)

4.3.1 Task Frames for Asyncs

According to the help-first policy in work-stealing, the tasks defined by async's are to be made available for idle workers to steal. Hence they have to be wrapped in a data structure that carries all the information that is needed to execute the async. This includes the value of all the local variables that are used in the async at the point when the async was created, the body of the async and the *Immediately Enclosing Finish* (IEF) that this task must report to on completion. Thus, we create a new frame, called the *TaskFrame* that acts as a wrapper for all this information.

```
void fibSlow (Worker w, Frame f) {
    fibFrame frame = (fibFrame) f;
    int n = frame.n;
    BoxInteger x = frame.x;
    BoxInteger y = frame.y;
    BoxInteger z = frame.z;
    switch (frame.pc) {
    case 0:
        if (n < 2) {
            z.val = n;
            return;
        }
        final BoxInteger x = new BoxInteger();
        final BoxInteger y = new BoxInteger();
        w.startFinish();

        TaskFrame tFrame1 = new Async1TaskFrame(x, n);
        w.pushTaskFrame(tFrame1);

        TaskFrame tFrame2 = new Async2TaskFrame(y, n);
        w.pushTaskFrame(tFrame2);

        frame.x = x;
        frame.y = y;
        frame.z = z;
        frame.n = n;
        frame.pc = 1;
        w.stopFinishSlow();
        if (w.popFrame()) {
            return;
        }
    case 1;
        z.val = x.val + y.val;
    }
}
```

Figure 4.10: Slow clone for the fib function for help-first policy work-stealing (unoptimized)

```
public class Async1TaskFrame extends TaskFrame {
    BoxInteger x;
    int n;

    public void execute() {
        fibFast (x, n-1);
    }
}
```

Figure 4.11: Task Frame for First Async in the fib example from Figure 4.2

A new type of *TaskFrame* has to be created for every async such that it holds all the information corresponding to this particular async. These *TaskFrame*s must be instantiated once for every instance of an async since each such instance represents a separate task that can be stolen. Figure 4.11 shows the *TaskFrame* corresponding to the first async in the *fib* example in Figure 4.2. This contains the local variables that are used by this particular async, x and n, and it is instantiated once for every execution instance of the first async.

Also since the new tasks generated by the asyncs are executed independently either by the worker that creates the tasks or by a different one in case it is stolen, it is better to "outline" the async into a separate function since then we can avoid jumps in to the async body and returns on end of it. This outlining transformation happens automatically in the front-end where the asyncs are implemented as inner classes with their body in a separate method of the class. Note that, in the work-first case, the "outlined" async is "inlined" into the body of the parent method. The *execute* method in the *TaskFrame* in Figure 4.11 is an example of the async being outlined in to a separate method. An important point to be noted is that the evaluation of the actual arguments to the function *fibFast*, which are part of the async body, also occurs inside this method.

4.3.2 Fewer Continuation points

The number of continuation points in each function is smaller in the case of help-first work-stealing as compared to that for work-first work-stealing. As mentioned earlier, the continuation points in the help-first case are the end points of finishes only. There are no continuation points after asyncs since the code following an async is executed immediately after making the async available for stealing. This is clearly seen in the slow clone of *fib* for help-first work-stealing in Figure 4.10, where there is

only one continuation point at the end of the finish. Though the slow clone in this example does not have much work to do after the continuation, in general, there can be any number of finishes in the same function in which case there will be more useful work after the continuation points.

4.3.3 Need to maintain the activation frame

Though we have a *TaskFrame* for each async, we also need another *frame* data structure like the one used for work-first work-stealing technique in order to store the activation frame of the functions. We call this a *ContinuationFrame*. The *ContinuationFrame* in addition to holding the local variables that will be used in the continuation of the function, also contains the program counter which indicates the continuation point the execution should resume from. It also holds links to the *ContinuationFrame*s of the parent functions if this function had been called sequentially. This *ContinuationFrame* is exactly the same as the one used for the work-first case. One such *ContinuationFrame* is created for every function. There is no need for the *ContinuationFrame* in a function if that function does not contain any continuation points, which in this case include the end point of finishes and the program points after a sequential call to a potentially parallel function.

4.4 Summary of Performance Results

This section summarizes the performance results comparing the work-first and help-first work-stealing scheduling techniques with the work-sharing techniques, as presented in [23]. All these results were obtained with the compiler support described in this chapter. They also serve as the baseline for the optimizations described in the next chapter.

The performance results were obtained on a 64-thread 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory; the runs on this machine were performed using Sun's Hotspot VM for Java version 1.6. All results were obtained using the `-Xmx2000M -Xms2000M` JVM options to limit the heap size to 2GB, thereby ensuring that the memory requirement for our experiments was well below the available memory on all the machines. All performance measurements followed the "Best of 30 runs" methodology proposed in [22] for Java runtime environments. The performance results are reported for eight Java Grande Forum (JGF) benchmarks, two NAS Parallel Benchmarks (NPB). The JGF and NPB benchmarks are more representative of iterative parallel algorithms rather than recursive divide-and-conquer algorithms that have been used in past work to evaluate work-stealing schedulers. All JGF experiments were run with the largest data size provided for each benchmark except for the Series benchmark for which Size B was used instead of Size C. There are five available sizes for the NPB benchmarks (S, W, A, B, C), and we used the intermediate Size A for all runs.

Figure 4.12 shows the performance of the JGF and NPB benchmarks. For convenience, the speedup measurements are normalized with respect to a single-processor execution of the X10 work-sharing scheduler [3]. We observe that on average, work-stealing schedulers outperform the work-sharing scheduler and the help-first policy performs better than work-first policy for work-stealing. This is because the JGF and NPB benchmarks contain large amounts of loop-based parallelism. For the `SOR`, `CG`, and `LUFact` benchmarks, the help-first policy outperforms the work-first policy by a large margin. In summary, the performance results show significant improvements (up to 22.8× on a 64-thread UltraSPARC T2) for our work-stealing scheduler compared to the existing work-sharing scheduler for X10.

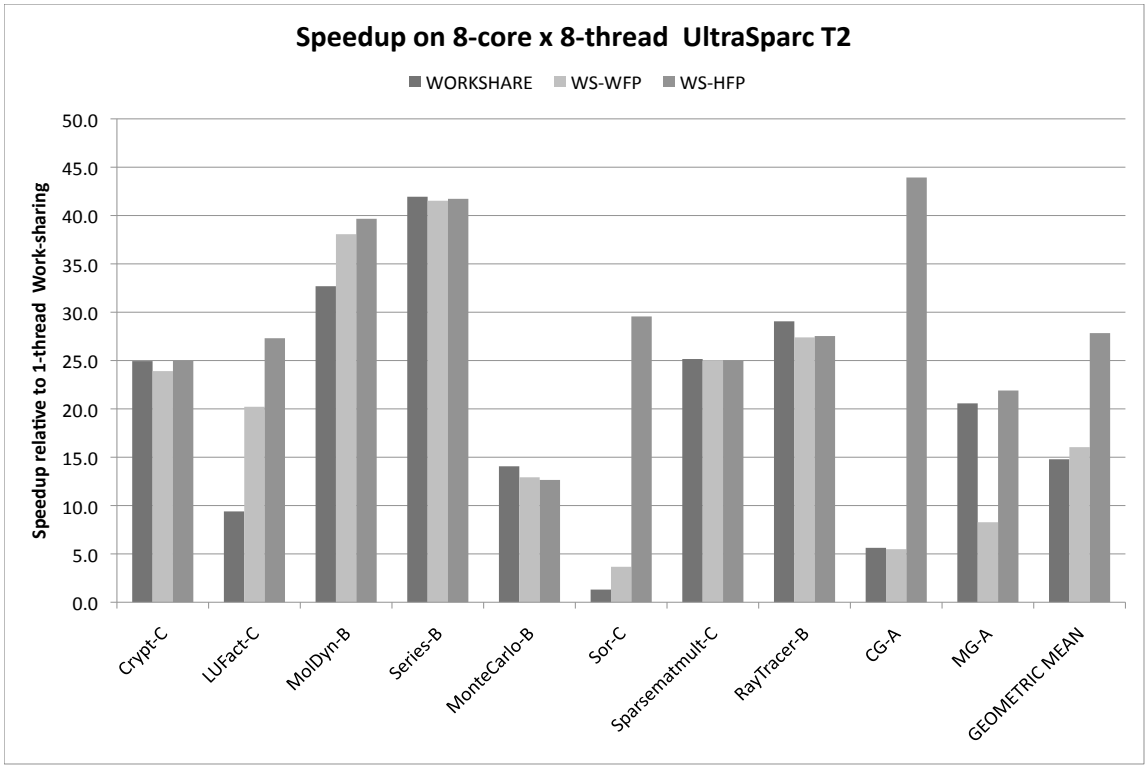


Figure 4.12: Comparison of work-sharing, HFP (Help-First Policy) and WFP (Work-First Policy) on a 64-thread UltraSparc T2 SMP.

Chapter 5

Optimizations in Work-Stealing Compilation

Compilation for work-stealing parallel runtime systems transforms parallel constructs like finish and async into equivalent code that implements the parallel semantics of these constructs atop the underlying runtime system. These transformations are usually performed separately for every parallel construct, without taking their context into account. We observe that there are significant opportunities for optimization. Our hope is that future optimizations, including the ones listed in this chapter, will greatly reduce the redundancies introduced by current compilation techniques for work-stealing runtimes, and that their relevance will increase as more applications are enabled for multicore execution.

This chapter describes some of these optimizations that can be performed on the code transformed for work-stealing parallel runtimes. Section 5.1 discusses the problem of identifying redundant stores into the heap frame data structure. Section 5.2 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*. It also shows the transformations necessary to use these application objects as *TaskFrames*.

5.1 Frame-Store Optimizations

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 naive approach outlined in Section 4.1.2 dictates that all the local variables be stored in the *frame* before every continuation point. Hence this code to store local variables on to the frame is linear in the number of local variables, at every continuation point. This can be a significant overhead in code size for large methods. We have observed that not every local variable needs to be stored into the frame before every continuation point since some of them 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. Thus there are opportunities to eliminate the redundant stores by performing data-flow analysis on the transformed code. This section describes the different kinds of analysis that can be used to identify redundant stores. Section 5.1.1 describes the usage of Live Variables analysis to remove redundant frame-store statements. Section 5.1.2 explains how Available Expressions analysis is used to identify and eliminate redundant frame-store statements. Section 5.1.3 presents the experimental results analyzing this set of optimizations and Section 5.1.4 proposes some extensions.

```
    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);
    }
```

Figure 5.1: Example HJ code snippet

Figure 5.1 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 and hence there are no continuation points after calls to these functions. Note that functions *f2* and *f4* may still be potentially parallel. They do not affect our analyses, since they do not create any continuation points in function *foo*. For simplicity, we consider parameters as local variables defined at the beginning of the method.

Figure 5.2 gives a simplified version of the example code transformed for work-stealing with the work-first policy without any optimizations. This version just shows the frame-stores added without depicting other transformations since we confine our interests to frame-stores in these optimizations. Since there are continuation points after statements *S3* and *S5*, all the local variables in the method are stored on to the frame before these statements.

5.1.1 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, it need not be stored into the frame before that

```
    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);
```

Figure 5.2: Code transformed for work-stealing with the work-first policy (unoptimized)

continuation point. This can be modeled using Live Variables analysis, i.e., only those variables that are live beyond the continuation point need to be stored into the frame before that continuation. The frame-stores of all other variables are unnecessary and hence they are marked as useless by the analysis. All such frame-stores are removed by a later pass over the transformed code.

The data-flow equations used to perform the Live Variables analysis are shown in Equations 5.1 and 5.2. The set *UEUses(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 the basic block *b*.

$$LiveIn(b) = UEUses(b) \cup (LiveOut(b) - Kill(b)) \tag{5.1}$$

$$LiveOut(b) = \bigcup_{s \in succ(b)} (LiveIn(s)) \tag{5.2}$$

Figure 5.3 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 on to the frame before *S3*. Similarly both x and y are not live beyond *S5* and hence need not be stored before *S5*.

An important point to note here is that if the Live Variable analysis is performed on the transformed code, it may yield pessimistic results. This is because since the transformed code already contains the frame-store statements for all the local variables, Live Variable analysis will incorrectly identify some local variables as live even though they are not used in the actual code (due to uses of the variables in the frame-store statements). One way to handle this issue is to do this analysis on the non-transformed code. But this would mean doing the optimization during the transformation which is not usually preferred. So, in our approach, we modify the Live Variables analysis to ignore the frame-store statements. This essentially means that the set *UEUses(b)* used in Equation 5.1 will not include any uses from frame-store statements in the basic block b. This ensures that there are no incorrect reports of live variables due to the uses in frame-store statements.

Uninitialized Local Variables

Since our work-stealing transformation stores every local variable in to the frame before every continuation point, it introduces uninitialized access to some local vari-

```
    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);
```

Figure 5.3: Redundant stores marked by Liveness analysis

ables. Suppose there is a local variable 'a' that is not defined until after the continuation point 'p'. But our transformation stores 'a' in to the frame before the continuation point 'p' which leads to an uninitialized access. It is illegal in Java to access such uninitialized local variables, and the Java bytecode verifier signals an error when such uninitialized accesses are present.

Such uninitialized accesses introduced by frame-store statements are handled automatically by the Live Variables analysis. If there is an uninitialized access to a local variable 'a' in a frame-store statement, that essentially means no definition of 'a' reaches the frame-store statement. This in turn means that there can not be a use of 'a' before a definition in the continuation following the frame-store statement. This is because the use of 'a' will then be an uninitialized access in the input code which is not allowed according to Java's strict typing rules. Now, since there is no use of

'a' before a definition in the continuation, 'a' is not live at the frame-store statement. Hence, the Live Variables analysis will mark all the frame-store statements that contain uninitialized access to local variables as redundant. The frame-store statement of the local variable b before statement *S3* in the example shown in Figure 5.3 is an uninitialized access to b and it is marked redundant by the Live Variables Analysis.

5.1.2 Available Expressions Analysis

In addition to the frame-stores marked for removal by performing the Live Variables analysis, we noticed that there are more frame-stores that are redundant. These are the frame-stores of the local variables that have already been stored on to the frame before a continuation point along all paths that reach this frame-store and have not been modified since they were last stored. This case ensures that the previous store of this local variable updated the frame with its latest value and hence the current frame-store is redundant and can be removed.

This case requires the ability to find if a local variable is redefined after it was last stored on to the frame along any path. The ideal way to model this is to think of the use of a local variable in a frame-store statement as an expression that is computed in that statement. The expression in this case is trivial because it just contains the value of the variable. This expression gets killed whenever the variable is redefined. Now the availability of this trivial expression before a frame-store statement indicates that the variable has already been stored on to the frame and it has not be redefined along any path since it was stored.

The data-flow equation used to perform the Available Expressions analysis is given in Equation 5.3. 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

```
    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);
```

Figure 5.4: Redundant stores marked by Available Expressions analysis

"available" on entry to the basic block b.

$$Avail(b) = \bigcap_{p \in pred(b)} (DEExpr(p) \cup (Avail(p) - ExprKill(p))) \tag{5.3}$$

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. Hence the analysis only needs to involve trivial expressions involving local variables that are stored in to the frame. Figure 5.4 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 in this case, since there was a store of a in to the frame before statement *S3* and it has not been modified since the store. So the analysis identifies that the expression a is available at the frame-store point before *S5*, and marks the frame-store as redundant.

An important point to note here is that we again need to differentiate between the use of a variable in a frame-store statement and other uses of the same variable. This

is because we perform the Available Expressions analysis by considering the use of a local variable in a frame-store statement as a computation of the trivial expression involving that variable. But there could be other uses of the same local variable in the method which should not be considered as a computation of the expression. Hence we modify the Available Expressions analysis to consider only the frame-store statements for trivial expressions involving local variables. This essentially means that the set *DEExpr(b)* will contain only the trivial expressions involving the local variables that are stored on to the frame in the basic block b and are not subsequently defined in b.

5.1.3 Performance Analysis

We now evaluate the performance of the Frame-Store optimizations on the code transformed for work-stealing schedulers. Initially, we use the work-stealing runtime with the work-first policy to study the performance of these optimizations. We study the performance on a wide variety of benchmarks which include seven Java Grande Forum (JGF) benchmarks, two NAS Parallel Benchmarks (NPB), and the Fibonacci, N-Queens, Graph-Coloring, and Graph Spanning Tree micro-benchmarks. As mentioned earlier, the JGF and NPB benchmarks are more representative of iterative parallel algorithms. Fibonacci and N-Queens micro-benchmarks have been used in the past [21] 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 [16]. The performance results were obtained on a 64-thread 1.2 GHz UltraSPARC T2 (Niagara 2) using Sun's Hotspot VM for Java version 1.6.

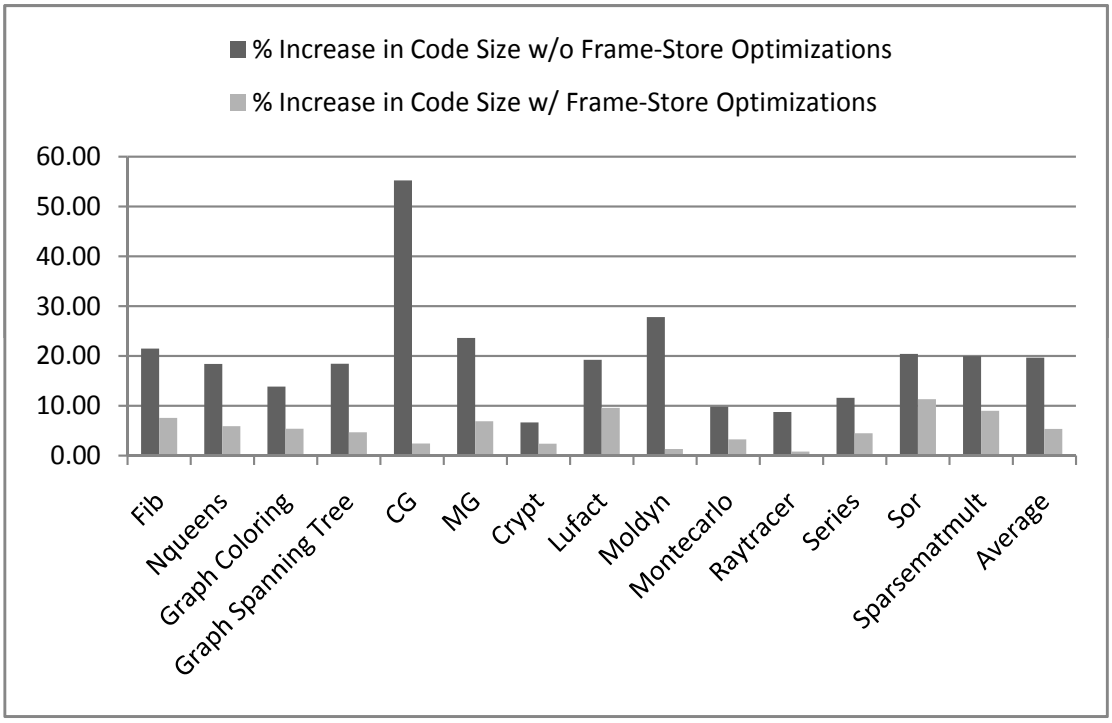


Figure 5.5: Increase in Code Size (in terms of the number of *Jimple* instructions) w/ and w/o Frame-Store Optimizations

Code Size with Frame-Store Optimizations

We first study 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 counted 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 counted the number of frame-store instructions that remained in these functions after Frame-Store optimizations. Figure 5.5 gives the percentage increase in code size of the potentially parallel functions in the benchmarks under study w/ and w/o Frame-Store optimizations. The reduction in code size that we have observed is as high as 52% for the potentially parallel functions of the 'CG' benchmark. On

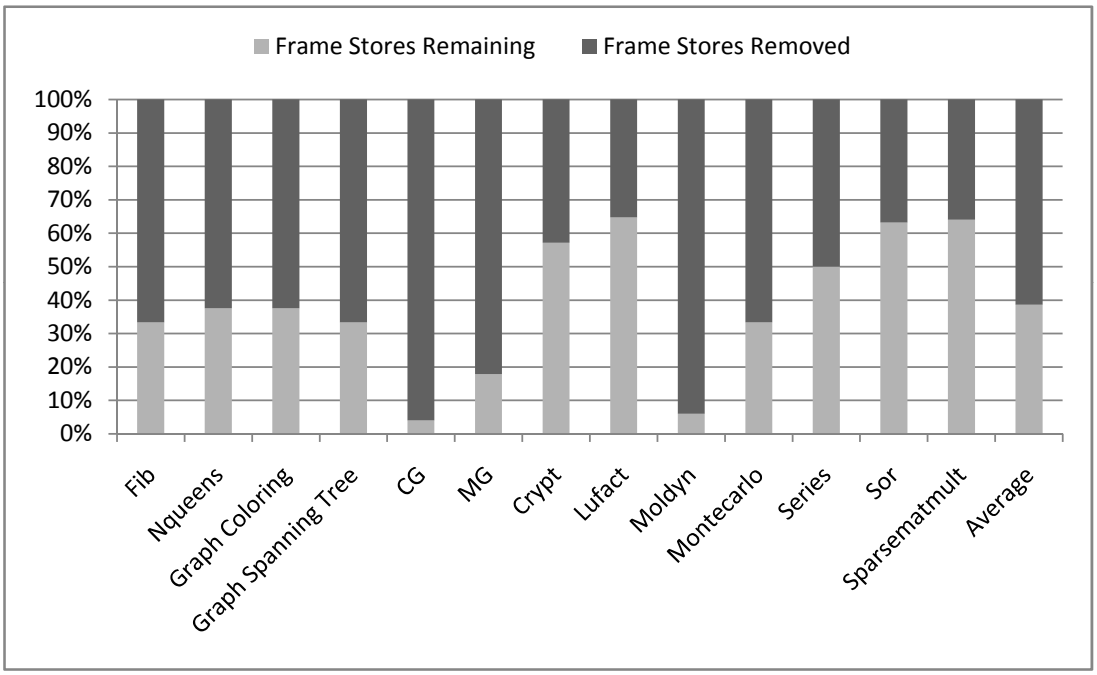


Figure 5.6: Dynamic counts of Frame-Stores instructions

average, there is a 14% reduction in size of the potentially parallel functions across all the benchmarks.

Dynamic Frame-Store Counts with Frame-Store Optimizations

We now show the dynamic counts of frame-store statements that were removed by this set of optimizations. The benchmark codes were first transformed to target work-stealing with the work-first policy. The set of frame-store optimizations were then applied on the transformed code. In order to obtain the dynamic counts, the optimized code was instrumented to get the dynamic count of the number of frame-store statements removed by the optimizations and the number of frame-store statements that remain. The instrumented code was run for each of these benchmarks under 1-thread case.

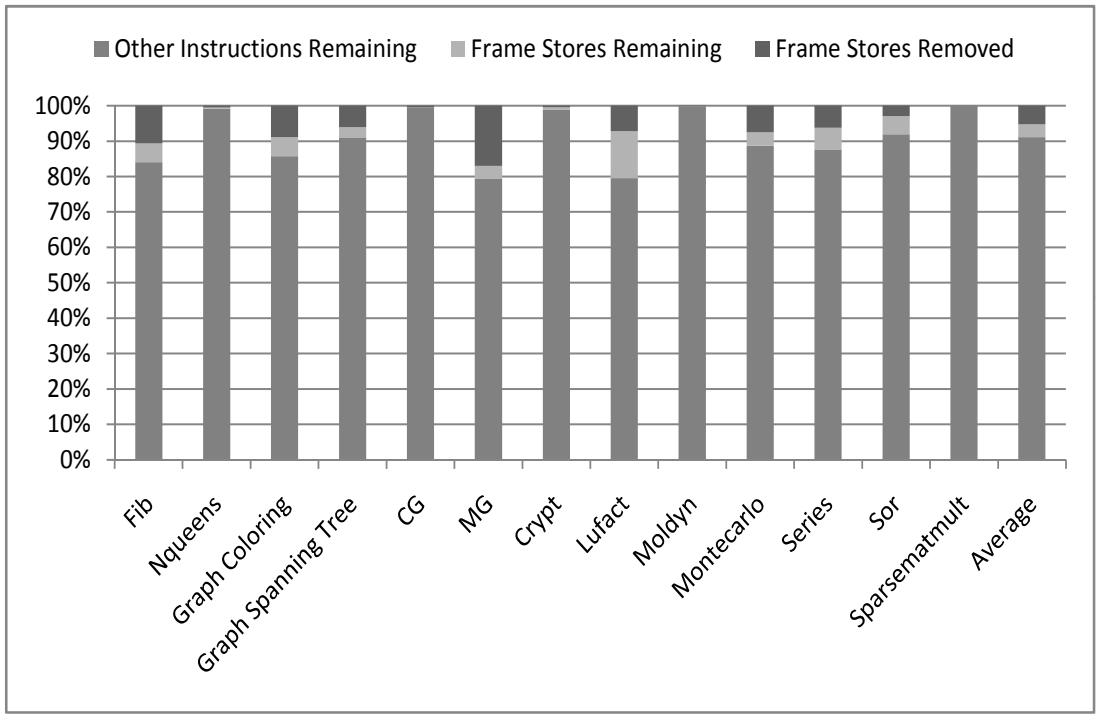


Figure 5.7: Dynamic counts of instructions

Figure 5.6 shows the percentage of frame-stores that were removed for each of the benchmarks. As evident from the graph, a significant number of frame-stores are being removed by the frame-store optimizations. On an average, about 60% of the dynamically executed frame-stores were removed. For some of the benchmarks like 'CG' and 'Moldyn', more than 90% of the dynamically executed frame-stores statements were removed.

We then turned our attention to study the significance of the dynamic counts of frame-store statements removed when compared to the total number of instructions[1] executed. Again, we instrumented the code to count the total number of bytecode instructions that were executed, other than the frame-store statements. Figure 5.7

[1]The terms 'statement' and 'instruction' are used interchangeably in this thesis. Both refer to a statement at the bytecode level.

gives the percentage of frame-stores removed with respect to the total number of instructions executed. On an average, the percentage of frame-store statements in the application is clearly low, i.e., around 10%, when compared to the total number of instructions executed. The percentage of frame-stores removed is thus even lower, around 6%. However, a significant percentage of instructions were removed for benchmarks like 'MG' and 'Fib'.

As mentioned earlier, the JGF and NPB benchmarks are representative of iterative parallel algorithms, i.e., the core of the algorithm is a parallel 'for' loop. These parallel 'for' loops could be chunked by the number of threads executing the benchmark [36]. But the versions used for this evaluation are unchunked, i.e., every iteration of the core loop is effectively an async. Hence the transformed versions of these benchmarks will include frame-store statements for every iteration of the loop. Even with these versions that have maximal frame-store statements, the percentage of the instructions that are frame-stores is just 10%. If the chunked versions of the benchmarks were used, the percentage of instructions that are frame-stores would be even smaller.

Execution Times with Frame-Store Optimizations

With the knowledge about the dynamic counts of the frame-stores removed with respect to the number of instructions executed, we compare the execution times of the optimized code with the unoptimized version. Figures 5.8 and 5.9 show the execution times of 'MG' and 'Fib' benchmarks running with threads varying from 1 to 64 on an ULTRASPARC T2. There is no measurable difference in the running times of the optimized and unoptimized versions for both the benchmarks. The same trend is seen with other benchmarks too. This clearly shows that the overhead due to the frame-stores is not a significant component of the execution time for these benchmarks, though 10% of the instructions executed are frame-store statements on an average.

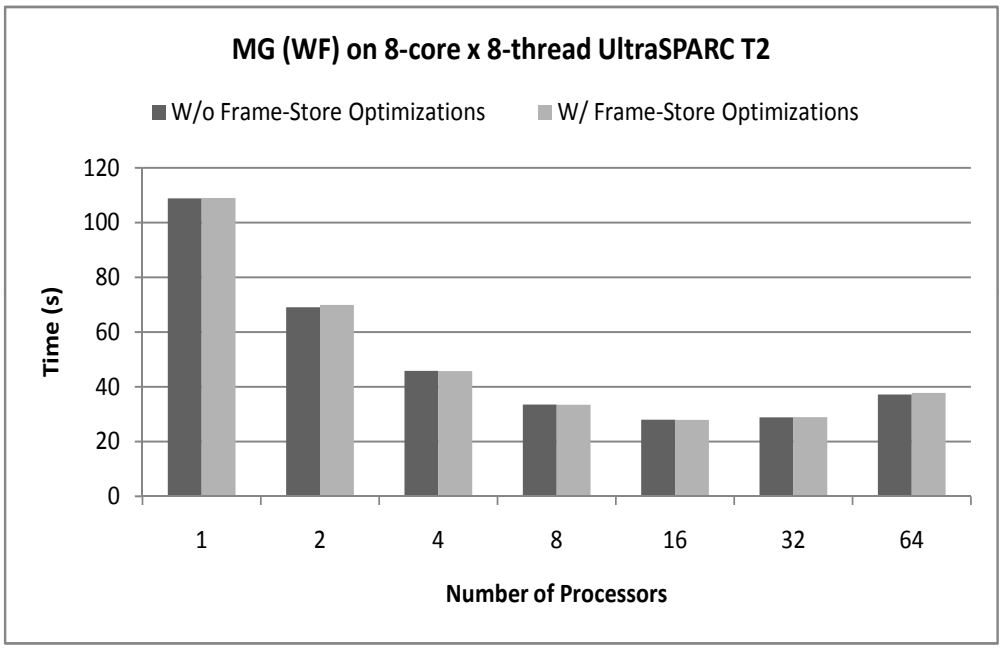


Figure 5.8: Execution Times for MG w/ and w/o Frame-Store Optimizations

These two benchmarks were selected for reporting since they showed promise with the highest percentage of frame-stores removed when compared with the total number of instructions executed.

5.1.4 Extending Frame-Store Optimizations

The Frame-Store Optimizations can be readily performed on the code transformed for work-stealing with the help-first policy. But, as mentioned earlier, the code transformed for the help-first policy has fewer continuation points as compared to the code for the work-first policy. Hence, the percentage of frame-stores that will be removed will be ever smaller. As we did not see any benefit even in the work-first case, we decided not to perform these optimizations for the help-first case.

There are more opportunities to remove frame-store statements by performing

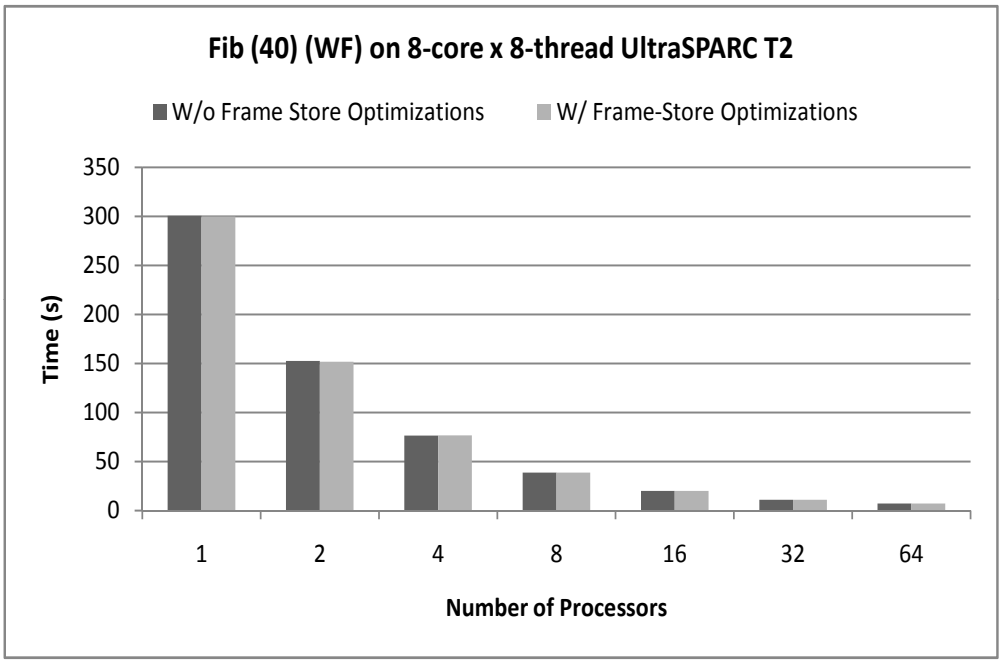


Figure 5.9: Execution Times for Fib w/ and w/o Frame-Store Optimizations

partial redundancy elimination. This will catch the cases when the frame-stores are redundant along some paths but not on others. In this case, the frame-stores will then be moved to the paths along which it is necessary thereby removing the redundancy along other paths. As we did not see any benefit in the execution times with the current set of optimizations and benchmarks, we decided not to pursue this further with partial redundancy elimination.

5.2 Using Application Objects as Frames

This section introduces a new optimization that uses the application objects as *TaskFrames* under the help-first policy of work-stealing. Section 5.2.1 describes the steps involved in identifying the objects that can be used as *TaskFrames* and the

transformations needed. Section 5.2.2 presents the study analyzing the performance of this optimization and Section 5.2.3 proposes some extensions to this optimization.

5.2.1 Objects-As-Frames Optimization

The code transformations for work-stealing with the help-first policy create *TaskFrames* that are used as a wrapper for the asyncs. A new type of *TaskFrame* is created for every async and they are instantiated once for every execution instance of an async. The *TaskFrames* contain information regarding the value of the local variables that are used in the async, the body of the async, and the *Immediately Enclosing Finish* (IEF). The local variables that are stored in *TaskFrames* may include some user-defined application objects. This optimization tries to use such application objects themselves as *TaskFrames*, thereby reducing the need for additional memory instantiations for the *TaskFrames*.

The challenge with this optimization is to find an application object that can be used as a *TaskFrame* for an async. Using an application object as a *TaskFrame* would involve extending the type of the object to include the fields and methods corresponding to a *TaskFrame*. Since a *TaskFrame* is specialized for a particular async, a type cannot be extended to represent more than one *TaskFrame*. This clearly dictates that 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 instantiated for every execution instance of an async, an object cannot be used as a *TaskFrame* for more than one execution instance of the async. In order to avoid this, we add a flag field to the object's type. We then use this flag to dynamically check if a particular object has already been used as a *TaskFrame*.

This 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 as the *TaskFrame* for this async. Now, 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 if an object has already been used as a *TaskFrame*. The point where the *TaskFrame* is created is now wrapped by a conditional to check the *used* flag of the object that is selected to be used as the *TaskFrame*. If the object has already been used, then a new *TaskFrame* is created. Otherwise, the object is used as the *TaskFrame* for this particular instance of the async. In this case, the remaining parameters of the async are stored on to the object.

Figure 5.10 shows the optimized version of the fast clone of the Fib example that was transformed for work-stealing with the help-first policy. Here the variable x which is a *BoxInteger* is being used as a *TaskFrame* for the first async. The modified version of *BoxInteger* is shown in Figure 5.11. It 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* case. 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 actually not needed in the fib example in Figure 5.10 because there is a new instance of the variable x being created for every execution instance of the first async in the method. But, in general, this may not be true. Also, if the variable that is promoted to a *TaskFrame* is a parameter to the method, there is no way to check if it has already been used as a *TaskFrame*. This is especially true for the Graph Spanning Tree example that is used to study the performance of this optimization later in Section 5.2.2. Hence, we stick to the use of the flag to overcome this problem. Also, note that the local variable y is not being used as a *TaskFrame*

```
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);

    TaskFrame tFrame2 = new Async2TaskFrame(y, n-2);
    w.pushTaskFrame(tFrame2);

    frame.x = x;
    frame.y = y;
    frame.z = z;
    frame.pc = 1;
    w.pushFrame(frame);
    w.stopFinishFast();
    if (w.popFrame()) {
        return;
    }

    z.val = x.val + y.val;
}
```

Figure 5.10: Fast clone for the fib function for help-first policy work-stealing optimized using application object as a frame

```
public class BoxInteger extends TaskFrame {
    int n;
    boolean used;

    public void execute() {
        fibFast (this, n-1);
    }
}
```

Figure 5.11: BoxInteger being used as a Task Frame for First Async in the fib example from Figure 5.10

for the second async in the fib example. This is because y is of type *BoxInteger* too and our restriction clearly says that a type cannot be promoted to a *TaskFrame* for two different asyncs.

Overheads due to this Optimization

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 in the application which are not being used as *TaskFrames*. These objects carry the overhead of the extra fields that are added to promote the type to a *TaskFrame*. There is also the additional boolean field, the *used* flag, that is added to every type when it is extended to a *TaskFrame*. This adds to the overhead of the objects of this modified type. Also, the optimization adds a conditional statement to test if an object has already been used as a *TaskFrame*. There is one conditional statement for every execution instance of an async that is optimized. This is one additional overhead that comes about because the choice of using an object as a *TaskFrame* is delayed until the execution time. This can be avoided if we can determine whether an object can be used as a *TaskFrame* for an async at compile time. We plan to address these issues as part of our future work.

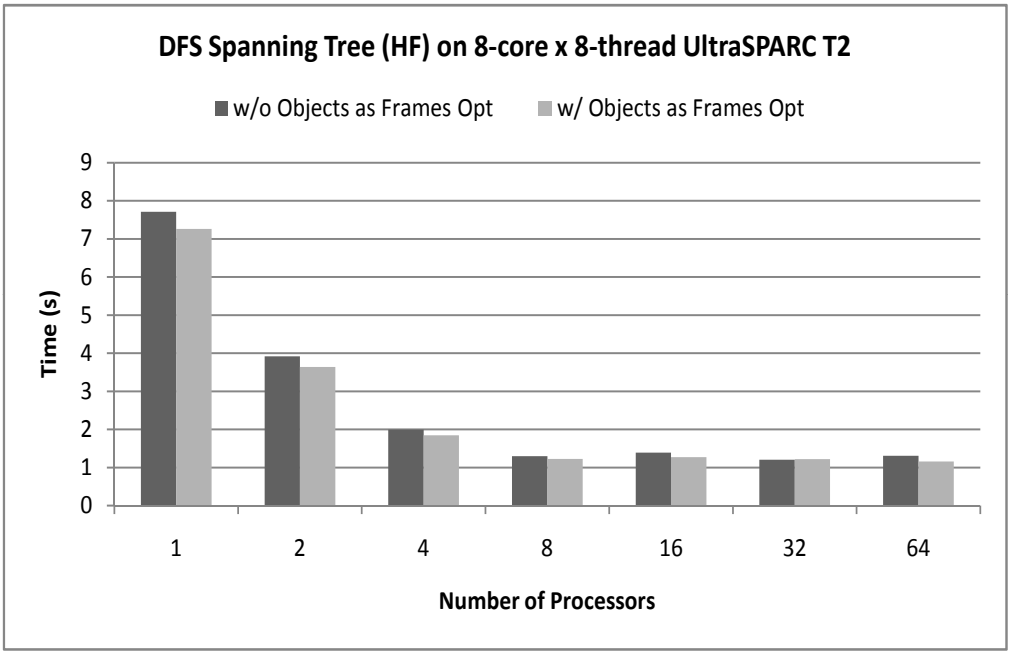


Figure 5.12: Objects-As-Frames Optimization on Graph Spanning Tree

5.2.2 Performance Analysis

We now use the Graph Spanning Tree benchmark to analyze 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* (DFS) on the graph. This benchmark was selected for evaluation since it has a nice structure that makes it a very good candidate for Objects-As-Frames Optimization. The code for this benchmark is given in Figure 3.4. As is evident from the code, there is 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 clearly suggests that the vertex object can be used as the *TaskFrame* for these asyncs. The optimization does exactly that.

Figure 5.12 shows the execution time of the optimized and the unoptimized ver-

sions 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. This is because, the overhead of the creation of new *TaskFrames* that were avoided by this optimization is linear in the number of vertices in the graph. This is the overhead that is seen under the 1-processor case. When the number of processors increase, this overhead is distributed among all the processors involved and hence there is lesser overhead involved with each processor. Also, as described earlier, the overheads due to this optimization catch up with the gains obtained by this optimization as the number of processors increase.

5.2.3 Extending Objects-As-Frames Optimization

The current version of this optimization targets the use of application objects in the place of *TaskFrames* under work-stealing with the help-first policy. However, this can be extended to do the same for the *ContinuationFrame* or the *Frame* as it is called under the work-first policy. The only difference in this case would be that the type of the object (that is being promoted to a *Frame*) should now be extended to contain fields for all the local variables that are used in the method from the continuation point. This could incur additional overhead since the number of local variables used in a continuation is, in general, greater than the number of locals used in an async. We plan to do this extension as part of our future work.

Chapter 6

Related Work

The three programming languages developed as part of the DARPA HPCS program (Chapel, Fortress, X10) 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 Cilk, OpenMP 3.0, Java Concurrency Utilities, Intel Thread Building Blocks, and Microsoft Task Parallel Library. Our results on efficient and scalable implementation of terminally-strict async-finish task parallelism can easily be integrated into any of the programming models listed above.

Work-stealing schedulers have a long history that includes *lazy task creation* [29] and the theoretical and implementation results from the MIT Cilk project. Blumofe et al. defined the fully-strict computation model and proposed a randomized work stealing scheduler with provable time and space bounds [8]. An implementation of this algorithm with compiler support for Cilk was presented in [21]. Agarwal et al. proved that terminally-strict parallel programs can be scheduled with a work-first policy so as to achieve the same time and space bounds as fully-strict programs [1]. To the best of our knowledge, our work presented in [23] 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[1].

The X10 Work Stealing framework (XWS) is a recently released library [16] that supports help-first scheduling 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 single-level-finish restriction leads to a control flow structure of alternating sequential and parallel regions as in OpenMP parallel regions, and enables the use of a simple and efficient global termination detection algorithm to implement each finish construct in the sequence. The library interface requires the user to provide code for saving and restoring local variables in the absence of compiler support. With these restrictions and an additional optimization for adaptive batching of tasks, the results in [16] show impressive speedups for solving large irregular graph problems. 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. An interesting direction for future research is to extend our compiler support to generate calls to the XWS library so as to enable performance comparisons for the same source code, and explore integration of the scheduling algorithms presented in this paper with the adaptive batching optimization from [16].

The Fork/Join framework [26] is a library-based approach for programmers to write divide-and-conquer programs. It uses a work-stealing scheduler that supports a help-first policy implemented using a variant of Cilk's THE protocol. In their approach, the thief needs to acquire a lock on the victim's deque when performing a steal in the Fork/Join framework.

[1]The recent Cilk++ release from Cilk Arts [15] allows the same function to be spawned and called sequentially from different contexts.

Chapter 7

Conclusions and Future Work

Though work-stealing has received a lot of attention in the recent past, the target of work-stealing has been the restricted set of fully-strict computations. Our joint effort [23] extended work-stealing for terminally strict computations and also introduced an alternate approach to work-stealing: the help-first policy. In this thesis, we discussed in detail the compilation techniques needed to support work-stealing along with some optimizations. We now summarize the contributions of this thesis and also propose extensions to be pursued as part of future work.

We gave a detailed description of the compilation support needed to schedule fully-strict computations characterized by Cilk's spawn-sync constructs using work-stealing. This included generating two clones, a fast and a slow clone, for each method that can spawn parallel tasks. The activation stack of the functions was stored in a heap *frame* data structure so that the thief gets the correct values of local variables to start the continuation with.

We extended these compilation techniques to support a broader class of computations known as terminally strict computations generated by HJ's async-finish constructs. We proposed different approaches to handle the challenges involved with

terminally strict computations. In order to support sequential calls to parallel functions, we extended the heap *frame* data structure to maintain the complete call stack. We also performed an inter-procedural analysis by building the call-graph to identify potentially parallel functions. We supported 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, like phasers.

We also described in detail the compilation strategies needed to support work-stealing by help-first policy. This involved creation of a new frame, known as a *TaskFrame* that acts as a wrapper for tasks created with asyncs. We also discussed how the continuations in a function differ in the case of help-first policy. Our results showed that there is a performance improvement of up to 22.8× on a 64-way SMP for our work-stealing scheduler compared to X10's work-sharing scheduler. We also observed that on average the help-first policy performs better than the work-first policy for work-stealing. Currently, the compilation targets either the work-first or the help-first policy work-stealing. A topic for future work is to have the compiler automatically decide if a task must be scheduled under work-first or help-first policy.

We performed an optimization to remove both the useless and redundant frame-store statements that were introduced to store the local variables in to the frame by these transformations. We described the different kind of data-flow analyses needed to identify the redundant frame-store statements. We showed that, on an average, there is 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'. Though we were able to remove a significant percentage of the dynamically executed frame-store statements, 60% on an average, we did not see any benefit in the execution times since the overall overhead due to frame-store statements is low on current systems. This does not necessarily

mean that these optimizations are of no use. If the time to access memory increases compared to processor cycles and the memory bandwidth per core decreases, as is the trend now, we could see more benefits with the removal of frame-store statements in future than what our current results show.

The other optimization we described was that of using the application objects as *TaskFrames* in the case of work-stealing with the help-first policy. This aimed at reducing the cost of instantiating new *TaskFrames* for every async. Our results showed 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 delayed till execution time. A topic for future work is to extend this optimization so that we could decide and replace the use of *TaskFrame* with an application object, at compile time. Another future work topic is to extend this optimization to identify application objects that could replace the *Frames* used under the work-first policy and the *ContinuationFrames* used under the help-first policy.

Bibliography

[1] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. 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] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, Apr. 2005.

[3] R. Barik, V. Cave, C. Donawa, A. Kielstra, I. Peshansky, and V. 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] R. D. Blumofe. *Executing multithreaded programs efficiently.* PhD thesis, Cambridge, MA, USA, 1995.

[5] R. D. Blumofe, Charles, C. E. Leiserson, and S. J. C. C. Space-efficient scheduling of multithreaded computations. In *SIAM Journal on Computing*, pages 362–371, 1998.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[9] A. R. Board. *OpenMP Fortran Application Program Interface v 1.0*, 1997.

[10] A. R. Board. *OpenMP Fortran Application Program Interface v 3.0*, 2008.

[11] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[12] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[13] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[14] 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 *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.

[15] Cilk Arts. *Cilk++ Programmer's Guide Version 1.0.2*.

[16] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. 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.

[17] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 Object-Oriented Programming, 9th European Conference*, 1995.

[18] M. F. Fern'andez. 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.

[19] R. Finkel and U. Manber. DIB—a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2):235–256, 1987.

[20] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *In First Symposium on Operating Systems Design and Implementation*, pages 201–213, 1994.

[21] M. Frigo, C. E. Leiserson, and K. 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.

[22] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[23] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium (To Appear)*, 2009.

[24] IBM. *X10: The New Concurrent Programming Language for Multicore and Petascale Computing.*

[25] R. H. H. Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17, New York, NY, USA, 1984. ACM.

[26] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[27] I.-T. A. Lee. The JCilk Multithreaded Language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition.* Addison-Wesley, 1999.

[29] E. Mohr, D. A. Kranz, and R. 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.

[30] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[31] T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[32] D. M. Peixotto and R. Barik. Implementing work-stealing with lazy frame-copying. Private Communication, 2008.

[33] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[34] Rice University. *Habanero Multicore Software Research project*.

[35] J. Shirako, D. M. Peixotto, V. Sarkar, and W. 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.

[36] J. Shirako, J. Zhao, V. Nandivada, and V. Sarkar. Chunking Parallel Loops in the Presence of Synchronization. In *ICS '09: International Conference on Supercomputing (To Appear)*, 2009.

[37] X. Teruel, P. Unnikrishnan, X. Martorell, E. Ayguadé, R. Silvera, G. Zhang, and E. Tiotto. Openmp tasks in ibm xl compilers. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 207–221, New York, NY, USA, 2008. ACM.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. 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.