Efficient and Scalable Implementation of an Object-Oriented Multithreaded Language

Seiji Umatani

Efficient and Scalable Implementation of an Object-Oriented Multithreaded Language

by
Seiji Umatani

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Informatics
in the Graduate School of Informatics.

Kyoto University
January, 2004

Abstract

This thesis proposes various techniques for implementing a modern multithreaded language OPA, which is an extended Java programming language that supports object-oriented programming and exception handling. For object-oriented parallel computing as in Java, each thread needs to keep its identity to implement the `synchronized` construct and each thread should have ability for general synchronization (suspension and resumption) to realize mutually-exclusive access to shared objects. For elegant exception handling, OPA employs a *join* construct with dynamic scope which enables an exception handler to catch an exception thrown by any of the child threads during parallel execution. For efficient implementation of multithreaded languages, *laziness* is an important idea; for example, Lazy Task Creation (LTC) is a well known technique for good load balancing. In this thesis, we pursue laziness for the modern language features, including thread identity preservation, general synchronization, and dynamically-scoped join. Also, the OPA system generates C code for good portability; this makes the adoption of LTC difficult. Although the implementation of the Cilk language has already overcome this difficulty in limited (well-structured) multithreaded computations, our implementation not only adopts LTC but also supports the modern language features and furthermore achieves better performance than Cilk.

OPA has loop constructs, and allows thread creation during a loop iteration. If we use the usual implementation of LTC for loops, we cannot divide the iterations into the uniformly sized works among the processors and the number of task creations will become large. In order to obtain the efficient load-balancing of LTC even in such cases, we propose an extension of LTC suitable for iterative thread creation.

We propose an efficient and portable implementation scheme of exception handling for fine-grained multithreaded programming languages, and evaluate its performance. We eliminate overhead for exception checks by unifying suspension checks and exception checks. Since the OPA system employs LTC, we also describe the implementation issues on such language systems. In fine-grained multithreaded programs, a lot of threads

are created, and the nesting of fork-join becomes deeper. Before handling an exception thrown in the course of parallel execution, it is desired to wait until all threads sharing the goal of the parallel execution finish (or abort) their execution. If we can abort execution of threads that can be aborted as soon as possible, useless computation is avoided and the total performance is improved. In this thesis we propose techniques for such case.

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my greatest supervisor, Professor Taiichi Yuasa, for his invaluable supports and directing me to this interesting research area.

I would also like to thank my Prof. Masahiro Yasugi, who has led the OPA project. He is great resources for discussing ideas and concerns. He is extremely helpful and willing to dedicate time to helping me.

I am also grateful to Dr. Tsuneyasu Komiya for his many insightful comments and supports, which really help me.

The OPA project has been a team effort and I am indebted to all the people who have contributed in some way to the OPA system.

I am also grateful to Professor Kei Hiraki at the University of Tokyo for allowing us to use Sun Ultra Enterprise 10000.

Finally, I would like to thank my family. Their support has always been important to me.

The research was supported in part by the 21st century COE program in Japan.

Chapter 1

Introduction

In this work, we have developed several efficient implementation techniques for a modern parallel language. We are developing an object-oriented parallel language OPA (an Object-oriented language for PArallel processing), and some of these techniques are used to efficiently implement OPA's constructs for thread creation and synchronization, which support a variety of parallel processing models including irregular computations using dynamically created (*forked*) threads.

The main advantage of our implementation techniques is that they do not interfere with other advanced features which most modern parallel languages are expected to provide, such as synchronization (or cooperation) of threads through objects, exception handling, and so on.

Among these features, in the OPA language, exception handling is designed so that it is suitable for OPA's parallel execution model. As a result, implementation issues for exception handling are closely related to the implementation techniques for thread creation and synchronization. Therefore, we have implemented exception handling in an efficient manner while the implementation of threads keeps high performance in our OPA system, and confirmed its high efficiency using several benchmarks.

This thesis provides the details of our implementation for parallel processing and exception handling.

1.1 Motivations

High-level programming languages for parallel processing are quite useful to develop reliable, reusable and efficient applications on various parallel architectures including shared-memory architecture and distributed-memory architecture. In high-level parallel programming languages for practical parallel processing, it is desired for the programmers to be able to describe practical parallel programs with irregular computation and/or side effect easily and safely as well as to be able to execute them on parallel computers efficiently. To support irregular computations, many practical languages have various features for runtime thread creation and exception handling.

The primary reason for the utilization of parallel programming languages is that we can expect our parallel program running on a parallel computer is much faster than the sequential version of the program running on a uniprocessor machine. However, this expectation sometimes results in disappointment because of the poor performance of parallel programs.

Poor performance can be caused by several factors. The degree of parallelism in the algorithms is one of the most important factors because it puts a strict upper bound on the performance achievable by the program. Some algorithms have a limited amount of parallelism and thus it is not possible to increase performance beyond a certain number of processors.

Another factor is the inefficiency of the language implementation. Even if the degree of parallelism is sufficient for scaling up with the number of processors, each parallel program executed on the inefficient implementation may have poor absolute performance when compared to a sequential program. That is, while the performance of the implementation executing parallel programs needs to be scaled up with the number of processors, the implementation must also take care about runtime overhead for parallel constructs to achieve good absolute performance.

Some previous studies have tackled this problem and their parallel language implementations achieve good absolute performance and relative speedup on parallel computers. But, some of these languages simplify parallel constructs for efficiency and, as a result, limit the types of parallel processing they support. Moreover, unfortunately,

most of these implementations do not assume the features other than their own parallel constructs. In particular, handling an exception during the parallel execution is not well-defined in these languages.

For this reason, when we implement a parallel language which supports modern features such as exception handling or object-based synchronization for flexible and irregular parallel processing, we cannot apply these previous techniques directly to our implementation.

In this thesis, we provide implementation techniques for our language OPA that support the above features. These techniques can be applied to other modern languages, and the result of this thesis will make it possible for programmers to use modern language features in their parallel programs without being worried about its runtime overhead.

1.2 Our Approach

Multithread constructs and their implementation should satisfy the following properties. First, they must be efficient. In other words, we must reduce the overhead of thread creation and synchronization. We would like to write fine-grained multithreaded programs using them. So, if they incur unacceptable overhead, most programmers would not use them and, instead, would create coarse-grained language-level threads h and manage fine-grained programmer-level threads explicitly. For this reason, we propose efficient implementation (compilation) techniques for OPA. Next, the language system should be portable. To achieve this, the OPA compiler generates standard C code. In contrast, some other systems exploit assembly-level techniques for efficiency. Finally, they must provide sufficient expressiveness. Some other languages achieve efficiency by limiting their expressiveness. For example, the fork-join constructs of Cilk[23, 7] employ "lexical-scope", and Cilk does not provide other types of synchronization. Cilk's constructs are simple but not flexible enough to write various irregular programs.

In this thesis, we propose three techniques which reduce overhead of fork-join style constructs. Their common key concept is *laziness*. Laziness means that we delay certain operations until their results become truly necessary. First, activation frames can be lazily allocated in the heap. In the case of implementing fine-grained multithreaded

languages in C, since each processor has only a single (or a limited number of) stack(s) for a number of threads, frames for multiple threads are generally allocated in the heap. For the purpose of efficiency, we allocate a frame at first in the stack, and it remains there until it prevents other thread's execution. Second, a *task* can be lazily created. In this thesis, a task is (a data structure of) a schedulable active entity that does not correspond to a single (language-level) thread. A blocked thread becomes a task to release the processor and the stack it uses. Also, for the purpose of load balancing, each thread may become a task to move around the processors. Our approach decreases the number of actually created tasks while it keeps good load balancing. Furthermore, we delay some other operations related to thread creation and, as a result, we can make the cost of thread creation close to zero. Third, data structures for synchronization can be created lazily. In OPA, such a data structure is necessary because synchronization points are dynamically-scoped[30] and each thread may be blocked.

In this thesis, we compare our efficient implementation of OPA with that of the Cilk language, which is a parallel extension of the C language. Cilk has fork-join style constructs, and its system achieves good dynamic load balancing on shared-memory multiprocessors. Since Cilk does not provide other types of synchronization such as communication (synchronization) through shared objects nor mutually exclusive method execution as in OPA, the implementation techniques of Cilk cannot directly be applied for OPA. However, by pursuing laziness, our OPA implementation obtains better performance than the Cilk implementation in spite of the richer expressiveness of OPA.

1.3 Organization of This Thesis

The rest of this thesis is organized as follows. Chapter 2 provides some necessary backgrounds about multithreading and exception handling. Chapter 3 overviews the OPA language. Chapter 4 describes our previous implementation of OPA. In Chapter 5, we propose lazy normalization schemes for the implementation of OPA. By "lazy normalization", we mean that we create a normal and heavy version of an entity (e.g., a heap-allocated frame) from a temporary and lightweight version of the entity (e.g., a stack-allocated frame) only when the normal one is truly necessary. In Chapter 6, we

extend our implementation techniques to support iterative thread creation. In Chapter 7, we present exception handling mechanism in our optimized OPA implementation. Chapter 8 presents related work and discusses expressiveness, efficiency, and portability. Chapter 9 shows benchmark results of some programs written in OPA and Cilk. Finally, in Chapter 10, we present concluding remarks.

Chapter 2

Background

Before discussing the design and the implementation of the OPA language, we briefly introduce and classify some kinds of parallel constructs and explain the syntax and the semantics of exception handling in this chapter. We also mention some issues on efficient implementation of these constructs.

2.1 Parallel Constructs

In this section, we discuss expressiveness of various constructs for the description of parallel computations in terms of (1) conciseness of the description and (2) describable types of parallel processing. Among various synchronization mechanisms of threads, we focus on synchronization mechanisms which wait for the completions of threads. We defer the discussion on the ease of handling exceptions until the next section. In our discussion, we use a programming language with the following assumptions:

- Constructs for parallel processing are explicit.
- Side-effects are permitted. (such as in C and Java)
- Fine granularity of parallel processing is permitted by language systems.

2.1.1 Description of Parallel Execution by Syntactic Constructs

Let us review the `forall` construct in terms of the two criteria. The `forall` construct can be used to describe parallel processing as follows:

`forall(i=1 to` N`)` *stat*

where N threads are created and they execute *stat* in parallel, and their completions are automatically synchronized. Concurrent Pascal-style `cobegin ... coend` construct can be used to specify a distinct statement for each thread as follows:

`cobegin` $stat_1$`;` $stat_2$`; ...` $stat_N$ `coend`

where N threads are created each executing the corresponding $stat_i$ and their completions are automatically synchronized.

The description by these constructs has the following features. (1) It is concise. (2) It only supports a simple type of fork-join parallelism and does not support a variety of parallel processing including irregular computations except for hierarchical structures that are formed by using `forall` or `cobegin ... coend` in *stat* recursively.

2.1.2 Thread Creation and Thread Synchronization by Operations

In order to describe a variety of parallel processing including irregular computations, a number of languages have been designed, in which operations for thread creation and thread synchronization (including operations on locations for synchronization) can be used. For example, in Java language [10], a thread can be created at runtime and various operations on the thread are supported. Here, we consider the following operations:

`thr = spawn` *stat*`;`

where a thread executing *stat* (*child* thread) is created and the reference to the child thread is obtained in the variable `thr`, and

`join(thr);`

where the completion of the child thread referred to by `thr` is waited for. The combination of these operations enables the following description of parallel processing where the number of created threads is not fixed:

```
{
  int i, n = 0;
  thread_t thr[N];
  for(i=0;i<N;i++)
    if(...) thr[n++] = spawn stat;
  barrier(thr, n);
}

void barrier(thread_t *thr, int num) {
  int i;
  for(i=0;i<num;i++) join(thr[i]);
}
```

where a thread executing *stat* is created only when the condition is met and the synchronization of the completion of every thread is expressed explicitly by the `join` statement.

Similarly, in some languages which employ data-flow synchronization (i.e., such synchronization that a thread which tries to extract the value from a location is suspended until the value of the location is determined), the following operations would be used:

```
ch = future exp;
```

where a thread evaluating *exp* in parallel is created and the reference to the location into which the result value will be stored is obtained in the variable `ch`, and:

```
val = touch(ch);
```

where the value stored in the location referred to by `ch` is extracted to `val` after the necessary suspension. The following description of parallel processing where the number of created threads is not fixed is similar to that using `spawn` and `join`:

```
{
  int i, n = 0, sum;
  int_channel ch[N];
```

```
  for(i=0;i<N;i++)
    if(...) ch[n++] = future exp;
  sum = reduce_add(ch, n);
}

int reduce_add(int_channel *ch, int num) {
  int r = 0;
  for(i=0;i<num;i++) r += touch(ch[i]);
}
```

where a thread evaluating *exp* is created only when the condition is met and the synchronization of the completion of every thread is expressed explicitly by the `touch` expression.

The description by these operations for thread creation (such as `spawn` and `future`) and thread synchronization (such as `join` and `touch`) has the following features. (1) It is not considered concise. In particular, the synchronization code (the loops with `join` or `touch` in the above examples) and thread management code (the array operations with `thr` or `ch` in the above examples) are required for the correct synchronization. The possibility of introducing bugs increases because of the too specific description. If the description of synchronization is incorrect, a serious symptom where the bug identification is difficult may be led by a thread that the programmer considers dead at some point while it continues its execution in practice. (2) It supports a variety of parallel processing including irregular computations.

In languages where thread creation and thread synchronization are described with explicit operations, the user cannot easily picture a configuration of the current parallel-execution context. This is because the synchronization point is not known until the synchronization operation is actually performed; such a synchronization point is the point where the result of the thread execution is necessary and should be known for the user to realize the goal why the thread is being executed.

2.1.3 Thread Synchronization by Syntactic Constructs

To reduce the description for thread management, a syntactic construct is useful. For example, in Cilk [23, 7] which is a parallel extension of the C language, the `cilk` construct

can be used to define a `cilk` function, which automatically manages threads created during the execution of the function body:

```
cilk void foo(...) {
  int i;
  for(i=0;i<N;i++)
    if(...) spawn funcall;
  sync;
  ...
}
```

where a thread executing *funcall* is created only when the condition is met and the synchronization of the completions of multiple threads is expressed explicitly by the `sync` statement. The threads created lexically within the body of the `cilk` function are automatically managed, and the `sync` statement expresses the synchronization of the completions of all threads which have been created before the `sync` statement is executed. In Cilk, thread creation is permitted only within `cilk` function bodies, and the same synchronization as the `sync` statement is implicitly performed when returning from `cilk` functions.

Compared to the description by operations for thread creation and thread synchronization, the description by the `cilk` construct and the `spawn` and `sync` operation has the following features. (1) It is more concise. Thread management code is eliminated and synchronization code is also reduced to a single `sync` statement. The possibility of introducing bugs decreases because of the shorter description length. (2) It supports a variety of parallel processing including irregular computations to some degree with the restriction that the programmer cannot directly specify threads involved in some synchronization and that one cannot allow a thread to survive across function-call boundaries which means that one cannot define an independent function to abstract several thread creations.

Next we consider a new syntactic construct `waitfor` (it corresponds to the `join` construct of the OPA language, see Chapter 3), which expresses both thread management (rather than by `cilk` function) and thread synchronization (rather than by `sync` operations) as follows:

`waitfor` *stat*;

where the completions of the threads created by `spawn` lexically within the body of `waitfor` (similar to the `cilk` function body) are synchronized. An example is as follows:

```
{
  int i;
  waitfor for(i=0;i<N;i++)
            if(...) spawn funcall;
}
```

Compared to the description in Cilk, the description by the `waitfor` construct has the following features. (1) It is more concise since `sync` operations are perfectly removed, and the possibility of introducing bugs decreases. (2) It adds the restriction that the programmer cannot change the synchronization point at runtime. (In Cilk, a `sync` statement may appear anywhere a statement is allowed in a Cilk procedure, including within the clause of an `if` statement and in other control constructs.)

2.2 Exception Handling

Exceptions provide a structured form of jump that may be used to exit a construct such as a block or a function (method) invocation. The name *exception* suggests that exceptions are originally designed to be used for exceptional operations. Exception handling is a basic mechanism that can be used to achieve the following effects: (1) jump out of a block or a function (method) invocation, (2) pass data as part of jump, and (3) return to a program point that was set up to continue the computation.

Exception mechanisms may be found in many modern programming languages, and every exception mechanism includes two constructs:

- an operation for throwing an exception, which aborts part of the current computation and causes a transfer of control,
- a handler mechanism, which allows a certain part of program code to be equipped with code to respond to exceptions thrown during its execution.

For parallel processing, we must extend the semantics of exception handling so that an exception during the parallel execution that cannot be handled by a certain thread can be properly handled.

When parallel programs are well-structured with thread synchronization by syntactic constructs, the extension of exception handling is obvious. For example, in the case of a `forall` statement, an exception that is thrown during the execution of a `forall` statement (including parallel execution of child threads) can be properly handled as the exception of the `forall` statement itself. We can define the language semantics so that, if a `forall` statement is wrapped by a `try-catch` statement, the exception thrown during the parallel execution can be caught by the exception handler of the `try-catch` statement and the whole parallel execution is stopped without using individual `stop` operations.

In the case of a `waitfor` statement, the extension is almost the same as for the `forall` statement: that is, an exception that is thrown during the execution of a `waitfor` statement can be properly handled as the exception of the `waitfor` statement itself.

On the other hand, thread synchronization by operations complicates the semantics of exception handling. The handling of an exception that is thrown during the execution of a child thread is not so obvious; that is, the way how the exception can be propagated outside the child thread is not trivial. In order to properly handle the exception, the language has to prepare operations for propagating the exception to the parent thread and for stopping a thread whose result is no longer needed, and the programmer has to describe the exception handling explicitly and carefully with the timing consideration.

In Java, operations for stopping multiple threads can be briefly described using a `ThreadGroup` object to manage related threads, but the operation itself cannot be omitted.

In some language designs [11], when a thread performs a `join` operation to another thread in which an exception is thrown, it receives the exception automatically, and when a thread performs the `touch` operation to a location to which an exception is propagated, it receives the exception automatically. For example, in the following code:

```
{
  int x, sum;
```

```
  int_channel ch1, ch2;
  ch1 = future f1();
  ch2 = future f2();
  x = f3();
  sum = x + touch(ch1) + touch(ch2);
}
```

an exception that is thrown by `f1()` can be handled as the exception of `touch(ch1)`. However, operations for stopping threads are still required (in this case, the thread executing `f2()` should be stopped) and also propagation of an exception is deferred until the corresponding synchronization operations are performed. Furthermore, if the language design employs explicit operations for storing a value to a synchronization location, the operation itself will not sometimes be executed due to an exception. For example, if `f1()` is defined as follows:

```
int f1() {
  int y;
  int_channel ch;
  y = g();
  ch = current_future();
  determine(ch, h());
  ...
}
```

the return value of `h()` is stored into `ch` explicitly. While an exception that is thrown during the execution of `h()` may be propagated to `ch`, an exception that is thrown during the execution of `g()` cannot be propagated to `ch`.

2.3 Implementation Issues

While a multithreaded language provides the programmers with a means to create and synchronize multiple threads, the implementation techniques for such a language automatically schedules these threads on processors of a parallel computer. To execute a multithreaded program efficiently, the scheduler must keep the processors busy and re-

duce interprocessor communication as much as possible in order to realize ideal parallel speed up.

Forked threads must be executed by the processors of a parallel machine in a manner consistent with the program-specified order, and in general, the assignment of threads to processors must be done at runtime. In many programs, threads are created only conditionally, and in these programs thread assignment cannot be determined until runtime. Furthermore, even if the threads can be statically identified at compile time, estimating the execution time of any given thread is not always possible; so load balancing should be considered in runtime scheduling. To overcome this problem, we must separate the language-level expression of parallelism in the program from the dynamic scheduling of threads at runtime. A multithreaded language permits this separation by incorporating a thread scheduler in its implementation.

The implementation of a typical multithreaded language automatically manages the low-level details of thread scheduling, and it does so with a "work-stealing" scheduler that is probably efficient. When writing a high-performance parallel application in such a language, the programmer can focus on expressing the parallelism in the algorithm independently of scheduling details with the knowledge that the implementation delivers high performance.

Beside automatic scheduling of the threads, in order to make the execution performance of fine-grained multithreaded programs comparative with sequential or coarse-grained multithreaded programs, the implementation techniques must reduce overhead related to the threads. The separation of expressing language-level threads and scheduling threads at runtime also allows the compiler to determine when costly operations are actually performed and when such operations are delayed.

The multithreaded language OPA and the compilation techniques presented in this thesis treat these implementation issues carefully to deliver high performance.

Chapter 3

OPA Language

This chapter presents an overview of the object-oriented parallel language OPA [32, 31]. Some of the key features of the language are the specification of parallelism and synchronization with fork-join constructs, irregular synchronization with object-based mutual exclusion, and exception handling.

In this chapter, we also describe *cactus stack*, which makes it easy to understand the behavior of fork-join type of parallel programs.

3.1 An Overview of the OPA Language

OPA (an Object-oriented language for PArallel processing) is a parallel extension of Java language [10]; we remove specifications on threads and monitors from Java and add new constructs for structured synchronization and relaxed mutual exclusion. Its design is intended to realize both ease-of-use and high performance of parallel processing. Note that we use Java as the base language because of its simple and clear semantics; our proposed scheme can also be applied to other object-based sequential languages.

OPA supports irregular parallelism with dynamically forked threads. In OPA, patterns of parallel processing can be divided into three types according to the relation among threads, namely fork-join parallel processing (Figure 3.1), cooperative parallel processing (Figure 3.2) and exclusive parallel processing (Figure 3.3). (They are described in detail later in this chapter.) In fork-join parallel processing, we divide a task

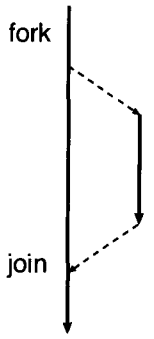


Figure 3.1: Fork-join parallel processing.

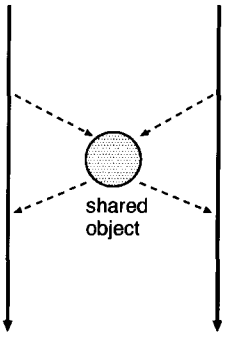


Figure 3.2: Cooperative parallel processing.

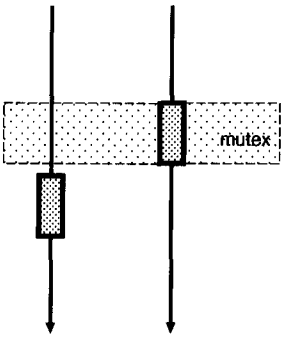


Figure 3.3: Mutually exclusive parallel processing.

into two or more subtasks; when a divided subtask can be executed concurrently, we can fork and join a new thread for the subtask in a structured manner.

Objects are mainly utilized to represent complex data structures, where an object can hold a reference to another object. In OPA, each object does not have a thread of control; as in sequential object-oriented languages, the method lookup and the subsequent method invocation on an object are performed by the thread that sends a message to the object. Mutual exclusion (serialization) is necessary for concurrent accesses (message passings) to an object to read/write the object's state consistently. In addition, objects are also used for cooperative parallel processing in which the related threads synchronize/communicate with each other in the course of their execution. The latter two synchronizations (cooperative and exclusive parallel processing) are recommended only when parallelism cannot be realized by the structured fork-join parallel processing.

3.2 Object-Based Synchronization

In this thesis, by synchronization, we mean that a thread suspends its execution until a resumption condition is satisfied. In OPA, most patterns of synchronization can be realized by the structured fork-join statements. However, other types of irregular parallel processing (cooperative and exclusive parallel processing) is necessary to describe

practical parallel programs. The support of such synchronizations in OPA makes its expressiveness more powerful than other parallel languages based on fork-join parallelism.

3.2.1 Exclusive Parallel Processing

In the object-oriented parallel computing, mutual exclusion (serialization) is necessary for concurrent accesses (message passings) to an object to read/write the object's state consistently.

In Java, a lock is associated with every object. But Java does not provide a way to perform lock and unlock actions separately; instead, their pairs are implicitly performed by the `synchronized` construct. We can use a `synchronized` statement as follows:

`synchronized(`*obj*`)` *stat*

The `synchronized` statement computes a reference *obj*; it then attempts to perform a lock action on that object on behalf of the current thread, and does not proceed further until the lock action has been successfully completed. After the lock action has been performed, *stat* is executed. If execution of *stat* is ever completed, either normally or abruptly (e.g., by throwing an exception), an unlock action is automatically performed on that object.

For programmers' convenience, a method may be declared `synchronized`; such a method behaves as if its body were contained in a synchronized statement. A `synchronized` statement (method) permits a single thread to lock an object more than once for avoiding unnecessary deadlocks.

OPA also provides the `synchronized` construct, but mutual exclusion realized by the `synchronized` construct can be relaxed by permitting simultaneous read-only accesses to the object, which results in the elimination of bottlenecks related to some objects accessed by many threads concurrently and frequently.

For this purpose, non-blocking read-only methods are used in OPA. A non-blocking read-only method may read the state of a mutable object but it can be executed without blocking. This is realized by updating the object state atomically. To incorporate non-blocking read-only methods, OPA uses relaxed mutual exclusion for the method defined with keyword `instant`. By defining an `instant` method with additional keyword

`readonly`, the programmers can specify the `instant` method as read-only (RO) typ All `instant` methods without `readonly` keyword are considered to be read-write (RV type.

Both RO and RW methods read the necessary variables of the object into loc variables atomically at the beginning of the method, while an RW method writes th values of local variables into the object atomically at the point where the rest of th method execution no longer updates the local variables. The compiler automaticall determines the update point with flow analysis. OPA also introduces a `vflush` statemer to specify the update point explicitly; the object is atomically updated when the `vflus` statement is executed.

The RO method can read the state of an object, even if an RW method is runnin on the object. The RO method is prohibited only from reading the state which is partl updated.

In contrast, the consistency control over multiple objects is not performed automat ically and must be specified explicitly.

In this way, the `instant` methods may be classified only at compile time. To in crease the number of RO methods further, OPA provides *dynamic (runtime) metho replacement*, which can be written as follows:

```
setmethod(m1, m2);
```

where `m1` and `m2` are method names. Once the above method replacement is performed on an object, `m1` messages sent to the object will be renamed to `m2`. In other words, when the object `obj`'s method is replaced, the message passing "`obj.m1(...)`" is executed as if it were "`obj.m2(...)`." By the dynamic method replacement, we can replace a RW method with a RO method. It eliminates some bottlenecks since the RO method can be executed with less strict mutual exclusion than the RW method.

This dynamic method replacement can also be used for cooperative parallel processing which we explain in the next section.

3.2.2 Cooperative Parallel Processing

In cooperative parallel processing, related threads synchronize/communicate with each other in the course of their execution. For instance, in producer-consumer relationship, the producer and the consumer communicate in a pipelined manner.

In OPA, objects can be employed for synchronization among cooperatively parallel threads which perform message passing to the shared objects. OPA provides useful classes for synchronization, such as I-store class (used for I-structure), FIFO queue class, and barrier synchronization class. An I-store object suspends all **get** accesses to it until a **put** access instantiates its value.

The above predefined classes for synchronization are written in the OPA language. For this purpose, OPA employs dynamic method replacement extended with the two special keyword **suspends** and `initial`. For example, when an I-store object is created, the following initialization is performed in the constructor:

```
setmethod(get, suspends);
```

Once **get** method is replaced to **suspends**, a thread which try to invoke **get** method must suspend its execution until the producer puts the value in it and then resets the replaced method in **put** method as follows:

```
setmethod(get, initial);
```

Java's approach for cooperative synchronization is different from OPA. Java employs *monitors*[15] with `wait()` operation (to enqueue the current thread in the object's waiting queue and unlock the object), `notify()` operation (to dequeue a thread from the object's waiting queue and make it runnable) and `notifyall()` operation (to perform `notify()` operation for all threads in the object's waiting queue).

However, the Java's approach has two problems. First, `wait()` operation is explicitly performed in the middle of a method, it is more difficult to keep the state of the object consistent. In OPA, every suspension occurs only before a method is invoked. Secondly, in Java, only one waiting queue is associated with an object. Therefore, to waken a specific thread prior to other threads waiting for their own conditions, we have to repeat the `wait()` operation until its waiting condition is satisfied and use `notifyall()` at

every point where any of the waiting conditions may become true. When the number of threads in the waiting queue increases, this scheme incurs significant overhead. In OPA, a waiting queue is associated with every replaceable method on an object.

3.3 Fork-Join Parallel Processing

OPA employs a `par` construct and a `join` construct instead of `spawn` and `waitfor` in Section 2.1.3. The rest of the thesis will use these `par` and `join` constructs. By attaching keyword `par` to a method call (or a statement), the execution of the method call (or the statement) is performed by a newly forked thread. By "`join` *statement*," *statement* is executed by the current thread and the completions of the new threads created during the execution of *statement* are joined with the completion of the `join` statement:

```
join {
  par obj1.m1();  // create a thread
  par obj2.m2();  // create a thread
} // synchronize the completions of the created threads
```

When a value calculated by a created thread is used for the rest of computation, components of a compound statement can be separated with a `join` label to indicate that the part before the `join` label is a join block:

```
{
  int x = par f1(n);
  int y = par f2(m);
join:
  z = x + y;
}
```

where the scope of the bindings of the variables (such as `x`, `y`) initialized by created threads is below the `join` label. The presence of the `join` label and the appropriate use of variables can be checked at compile time.

For synchronization, join targets (i.e. synchronizers) have dynamic scope in OPA. More precisely, we define that a synchronizer established by `join` is dynamically scoped.

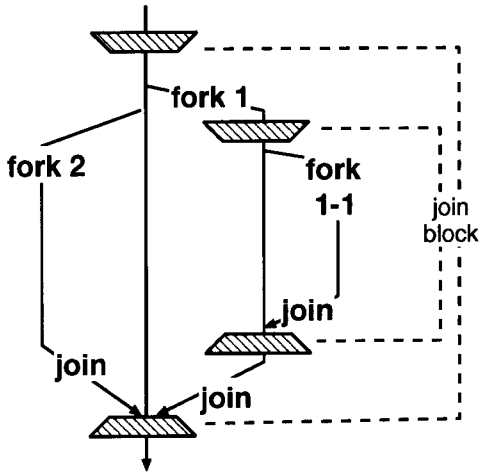


Figure 3.4: Structured synchronization.

Dynamic scope means indefinite scope and dynamic extent, where references to an established entity may occur anywhere and at any time in the interval between establishment of the entity and the explicit disestablishment of the entity.

For example, the execution of the following statement can be illustrated in Figure 3.4:

```
join {
  par f1();
  f();
}
```

where `f1` and `f` are defined as follows:

```
f1() { ...
  join {... par f1_1(); ...}
}
f() { ... par f2(); ... }
```

The join block in the figure represents the interval during which the body of `join` is executed, where `f1` has a nested `join` statement. The join target of `par f2()` executed in `f` is referred to using dynamic scope. The join target of `par f2()` will not change even if we replace `f()` with `par f()`. The same `join` construct is employed in COOL [2]

which is a parallel dialect of C++, where thread creation is performed by calling parallel functions (functions defined with keyword `parallel`); however, COOL does not involve exception handling.

3.4 Exception Handling

The syntax for exception handling in OPA is the same as in Java. We first explain the description and the meaning of exception handling in Java language. An exception can be thrown by a `throw` statement:

`throw` *exp*`;`

where the value of *exp* must be an object representing the exception. Only objects that are instances of the `Throwable` class (or of one of its subclasses) can be thrown by the Java `throw` statement. The exception stops the current execution and the control is transferred to the exception handler that is determined with dynamic scope.

The `try-catch-finally` construct is prepared for exception handling. Exception handlers for an exception thrown during the execution of a `try` block are described as `catch` clauses:

```
try {
  ... // an exception may be thrown.
} catch(Exception1 ex1) {
  ... // may be executed for Exception1
} catch(Exception2 ex2) {
  ... // may be executed for Exception2
} finally {
  ... // always executed
}
```

A `try` statement executes a `try` block. The `catch` clauses will not be executed when no exception is thrown. If an exception is thrown and the `try` statement has one or more `catch` clauses that can catch it, then control will be transferred to the first such

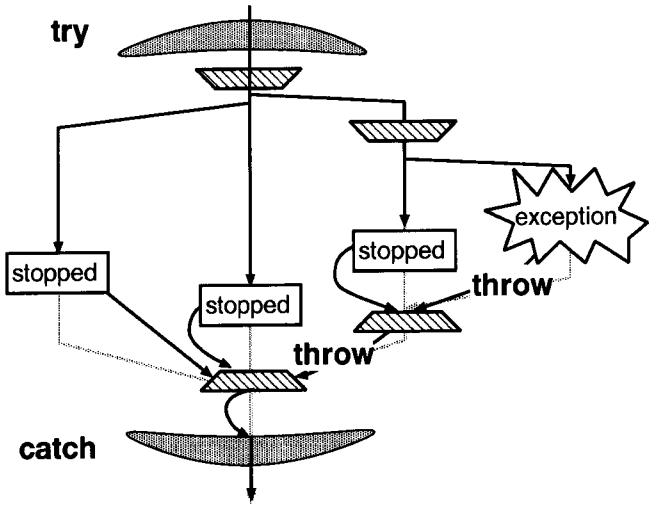


Figure 3.5: Handling an exception which is thrown during parallel execution.

`catch` clause. A `catch` clause can catch and handle an exception (object) of the specified class (or of one of its subclasses). If an exception is not caught, the exception will be propagated to `catch` clauses for the outer `try` block. The exception handler is referred to using dynamic scope; thus, the nesting of `try` blocks is dynamic: for example, if an exception is not caught in a method body, the exception is propagated to the calling point of the method.

If the `try` statement has a `finally` clause, then the `finally` block is executed, no matter whether the `try` block completes normally or abruptly (with an exception), and no matter whether a `catch` clause is first given control. If an exception is thrown during the execution of the `finally` block, the old exception (if any) is discarded.

An exception that is thrown during the execution of a `join` block (including parallel execution) can be properly handled as the exception of the `join` block itself. We can define the language semantics so that, if the `join` block is wrapped by a `try-catch` statement, the exception thrown during the parallel execution can be caught by the exception handler of the `try-catch` statement and the whole parallel execution is stopped without using individual `stop` or `abort` operations:

```
try {
  join {
    par f1();
    f();
  }
} catch( ... ) {
  ...
}
```

The execution of the above statement is illustrated as in Figure 3.5. If an exception cannot be handled by a thread, the exception is propagated to the join target of the thread, which then stops the other threads sharing the same join target. Thus, the description is simple and the handling of the exception during the parallel execution is obvious.

The thread which handles an exception has to execute necessary `finally` clauses before the control is transferred to a `catch` clause. The other threads that are stopped due to the exception have to execute necessary `finally` clauses before their termination. If a thread has acquired a lock for an object, the lock should be released as if the object is unlocked in a `finally` clause.

With an `instant` method, the update of the object's data is performed atomically at a single update point; therefore, if an exception is thrown before the update point, the object's data remains unchanged. To enforce the update before throwing an exception, we can use the `vflush` statement.

So far we have not discussed the complicated cases caused by multiple threads and `finally` clauses. Only one exception should survive if two or more exceptions reach to the same join target; and thus, one problem is how to determine the survivor. There is a similar case in the following sequential execution: if an exception is thrown during the execution of the finally block, the old exception (if any) is discarded in Java. However, there is no difference in execution order among parallel exceptions. One solution to this problem would be to give a priority to each exception. But, for simplicity, we decided that the survivor is the exception that reaches first. The other problem is how to precisely define the behavior of the stopped thread. It is an elegant idea to define the behavior as automatically throwing a special (non-user) exception `stopped` except for two cases:

when `stopped` is being thrown and when a finally block is being executed. Although the execution of a `finally` block is not stopped by other threads, the `finally` block itself may throw an exception, possibly discarding the `stopped` exception. OPA avoids this problem by defining that the `stopped` exception is automatically re-thrown.

3.5 Cactus Stack Model

In languages where thread creation and thread synchronization are described with syntactic constructs, the user can easily picture the configuration of the current parallel-execution context. This is because the synchronization point is known when a thread is created; such a synchronization point is the point where the result of the thread execution is necessary and is regarded as a goal why the thread is being executed. In sequential languages, an execution context forms a data structure, namely *stack*. Here we imagine an ideal control stack which only saves control transfer information and does not rely on an automatically-incremented program counter. The stack top holds the information about the statement to be executed. If the execution of a function body (consisting of statements) or a block body (consisting of (sub)statements) is required to execute the current statement (which is popped from the stack), those (sub)statements are pushed onto the stack. As such a stack, the user can picture the configuration of the current execution context or can know the goal of the current execution. On the other hand, the stack for parallel execution in Figure 3.4 would be a *cactus stack*, which changes as is shown in Figure 3.6. Every created thread has a goal to continue the work after the join; it has its own (sub)stack on a *join frame* which is used as a join target (Figure 3.6). The join frame returns its control only after all the stacks on the top of it become empty. The user can picture the configuration of the current parallel execution context as a cactus stack. Furthermore, an exception that is thrown during the parallel execution (Figure 3.5) makes the cactus stack change as is shown in Figure 3.7. These transitions are also straightforward to the users.

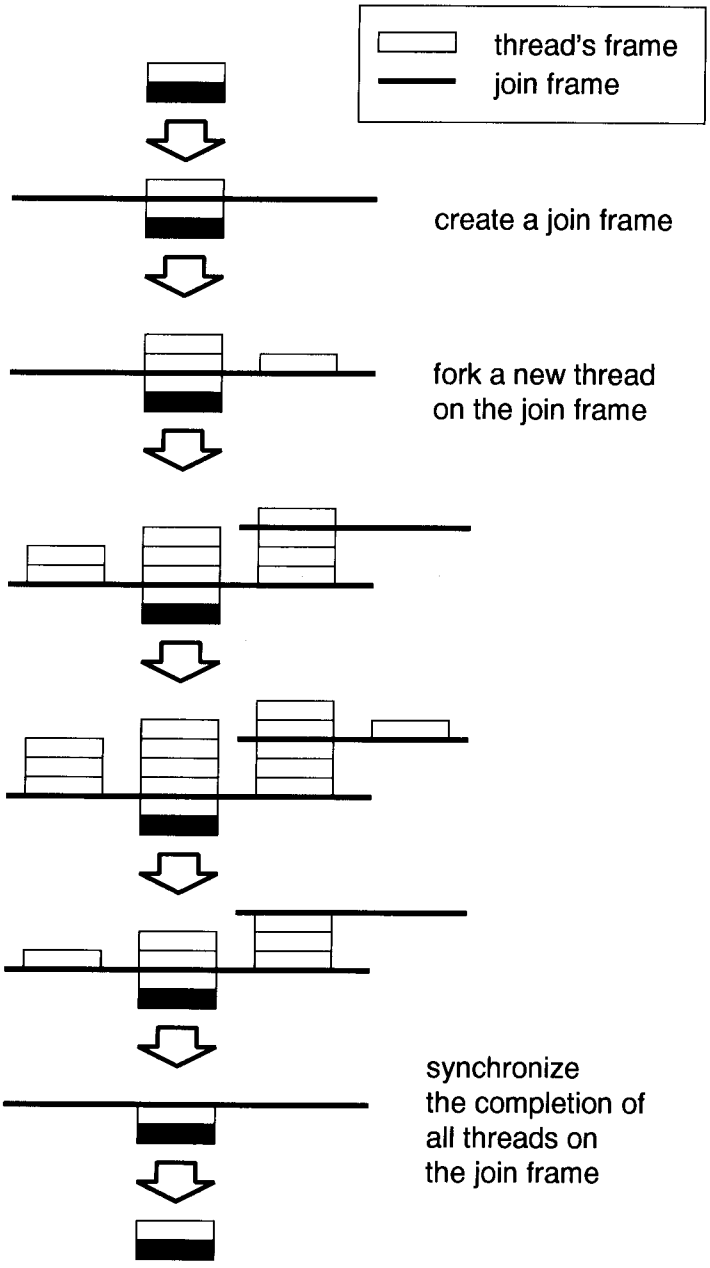


Figure 3.6: Transition of the cactus stack based on the structured synchronization.

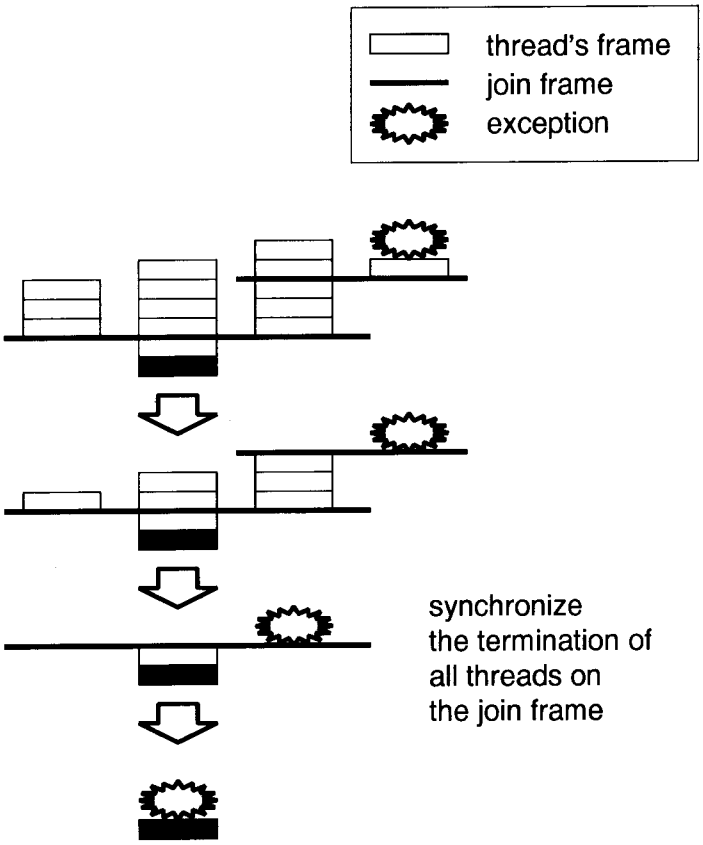


Figure 3.7: Transition of the cactus stack for an exception thrown during parallel processing.

```
 1 class Fib {
 2   private static int fib(int n) {
 3     if(n < 2) return n;
 4     else {
 5       int x = par fib(n-1);
 6       int y = fib(n-2);
 7     join:
 8       return x+y;
 9     }
10   }
11   public static void main(String[] args) {
12     int r = fib(36);
13   }
14 }
```

Figure 3.8: `fib` code in the OPA language.

3.6 Examples

Some examples are presented to see we can describe a variety of parallel programs easily and safely with a small set of language constructs, such as `join` and `par` constructs for synchronization, and `try-catch-finally` and `throw` constructs for exception handling.

3.6.1 Fibonacci Numbers

Figure 3.8 shows a simple Fibonacci program written in OPA. `par fib(n-1)` at line 5 means that its method invocation is executed concurrently by a newly created thread. Note that `fib` is a usual method; that is, we can also call it sequentially (line 6). This is desirable because a parent thread can participate in the computation and the number of thread creation can be reduced. In this program, the parent uses a `join` label, the variant of `join` synchronizers, for safely using the value returned by its child. This `join` label is required before the statement "`return x+y;`" to avoid the anomaly that would occur if `x` is used before it is computed.

3.6.2 Data Parallel Processing

To perform a parallel method call for each element of an array `objs` in a data-parallel manner, we can write the program as follows:

```
join {
  for(i=0; i<objs.length; i++)
    par objs[i].doit();
}
```

3.6.3 Search

To find two answers which are searched in parallel and stop the whole execution with an exception (the fact that the answers were found), we can write the program as in Figure 3.9.

```
 1 // main class  (to catch the exception)
 2 public class SearchStart {
 3   public static
 4    void main(String argv[]) {
 5     Table table = new Table();
 6     try {
 7       join {
 8         Node node = new Node(0);
 9         node.search(table);        // start parallel search
10       }
11       System.out.println("NotFound");
12     } catch(Found excp) {          // if solutions are found
13       System.out.println("Found");
14     }
15   }
16 }
17 // Node for search space
18 class Node {
19   int p;
20   Node(int p0) { p = p0; }
21   void search(Table table) throws Found {
22     ...
23     if (an answer is found) table.add(the answer);
24     else {
25       Node node1 = new Node(2*p+1),
26            node2 = new Node(2*p+2);
27       par node1.search(table);
28       par node2.search(table);
29     }
30   }
31 }
32
33 // Table for answers (throws an exception)
34 class Table {
35   int[] answers = new int[2];
36   int n = 0;
37   instant void add(int ans) throws Found {
38     answers[n++] = ans;
39     if (n >= 2) {
40       vflush;
41       throw new Found(this); // non-local exit with exception
42     }
43   }
44 }
```

Figure 3.9: search code in the OPA language.

Chapter 4

OPA Implementation

In this chapter, we explain several implementation techniques for our OPA language[33]. Among them, the implementation techniques for fork-join constructs described in this chapter will be improved in the next chapter.

Our OPA system consists of a source-to-source compiler from OPA to C and a runtime system written in C. Figure 4.1 illustrates the process by which an OPA program is compiled. OPA program files, which end with `.opa` by convention, are first translated into ordinary C code by the OPA compiler, producing `.c` files. The C code is then compiled using the C compiler and linked with the runtime system.

In this study, we pursue the high performance by minimizing the role of the runtime system and by performing some important operations for language-level thread management such as thread suspension/resumption in the generated C code.

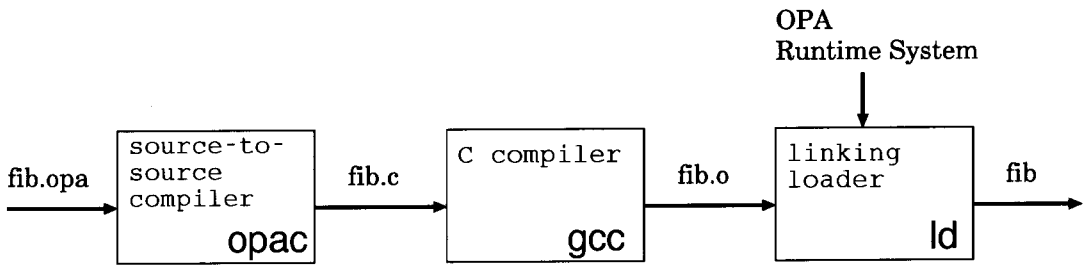


Figure 4.1: Compilation process of the OPA system.

4.1 Runtime Environment

First of all, we present global organization of the OPA runtime environment. As shown in Figure 4.2, each processor uses the following five data areas:

C stack This is used for C code which is generated by the OPA compiler. In the OPA system, a single OS-level thread is running on each processor, so each processor has only a single C stack.

Heap All objects created during the execution of an OPA program are allocated in this area. In OPA, each object is associated with a queue which contains threads (thread objects) waiting for acquiring the object's lock.

Frame area This area is managed with a free list of frames of a fixed size. Frames are used for several purposes: heap-allocated activation records, join frames, thread objects, and so on. In OPA, thread objects are semantically different from those in Java. Since thread objects are not visible from the programmers, they are allocated in this area for simplicity.

Ready Thread Queue This thread queue keeps pointers to thread objects that are ready for running on the processor. A suspended thread waiting for acquiring an object's lock is enqueued in the ready thread queue when the object is unlocked. Also, a suspended thread waiting for the completion of `join` synchronization is enqueued just after the synchronization. It is also used as a communication buffer for thread migration among processors.

Processor Specific Data Other processor specific data are arranged in a single data structure. For instance, it includes a field for method's return value, base addresses of the above areas.

4.2 Implementation of Join

When exiting from a join block, of course, the parent thread needs to wait for the completion of all the child threads created within the join block. When threads are

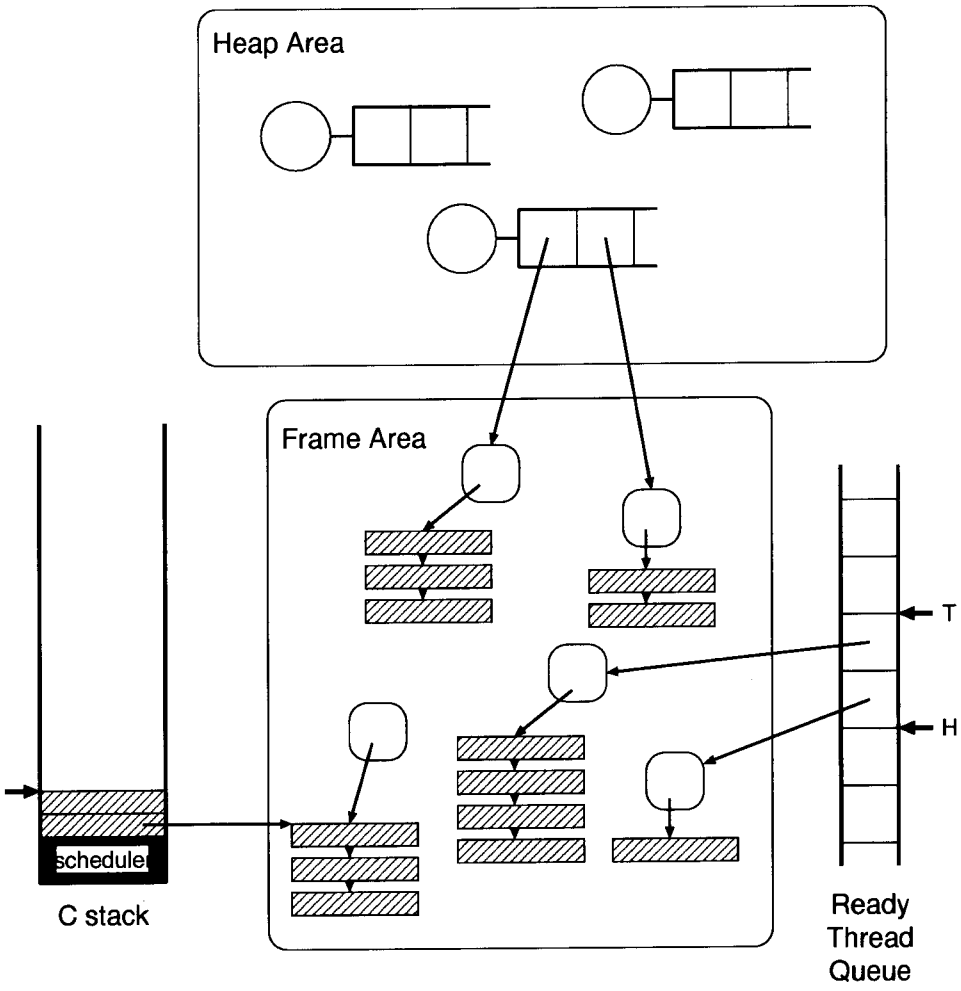


Figure 4.2: Organization of the OPA runtime environment.

created dynamically by operations, the number of child threads is, in general, known only at run-time. Thus, the OPA system prepares a counter for each join block which holds the number of child threads that has not yet joined to the `join` synchronizer.

Join synchronizers are implemented as *join frames*. The type of join frames is defined as follows:

```
typedef struct j_frame {
  int w; /* counter */
  int parent_jf; /* link to parent j_frame */
  int parent_jw; /* and its weight */
  f_frame *parent_fr; /* rendezvous */
}
```

The usage of each field is explained in detail in the following explanation of fork-join synchronization.

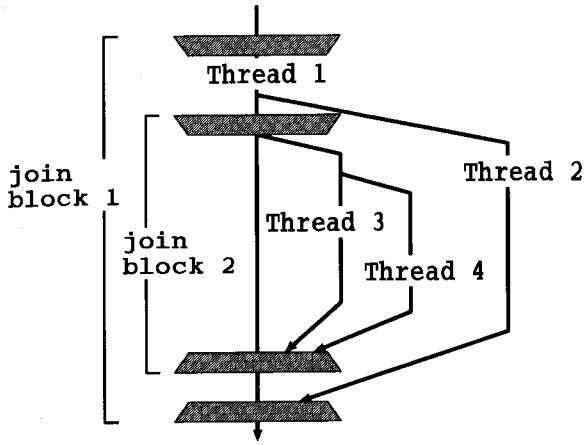
A join frame is used as a counter to hold the number of child threads and is allocated by a thread that enters a new join block. Each thread (more precisely, thread object) has the reference to the join frame which corresponds to its `join` synchronizer. If we use a simple counter for this purpose:

- When a new thread is created at runtime, the thread has the same reference as the parent thread since they synchronize at the same join point, and then it increments the counter.

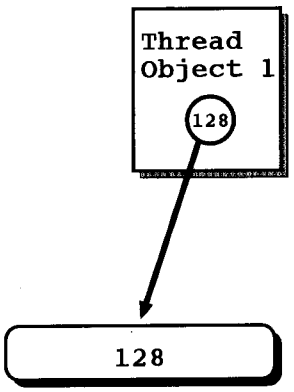
- At the completion of the thread execution, it decrements the counter.

Counter value 0 means that the entire synchronization has been completed.

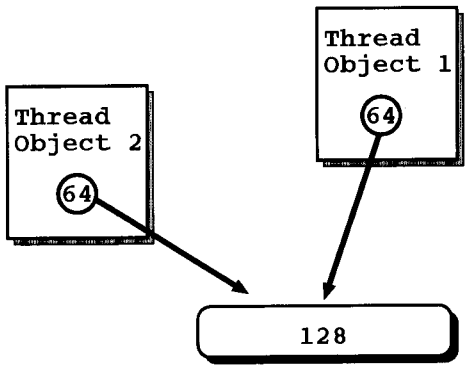
Using a simple counter, each thread needs mutually exclusive accesses to increment/decrement the counter. To avoid this overhead of each thread creation, we adopt weighted reference counting[1]. Figure 4.3 shows how join frames are allocated using weighted reference counts. The initial value of a counter is non-zero (in Figure 4.3, 128), and the thread that enters a new join block has the weight of the same value as the counter together with the reference to the join frame as illustrated in Figure 4.3 (b). We call them a *weighted reference*.



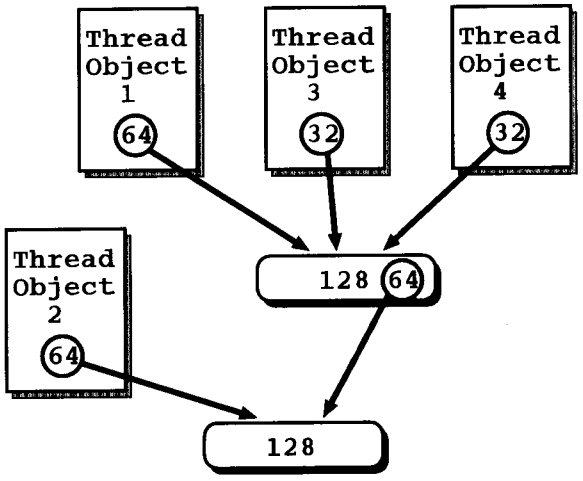
(a) Fork-join structure



(b) Creation of a new join frame



(c) The weight is split between two threads



(d) Creation of a nested join frame

Figure 4.3: Join frames using weighted reference counts.

- When a new thread is created at runtime, the thread has the weighted reference that refers to the same join frame as the parent thread. The weight of the parent thread is split between these two threads (Figure 4.3 (c)).

- At the completion of the thread execution, it subtracts its weight from the counter.

The benefit of this scheme is that creating a new child thread only needs the split of weight and no more operation and synchronization.

During the execution of a program, join blocks may be nested. For example, in the code of Figure 4.3, when thread 1 enters a new join block (join block 2), it is included in the join block 1 that was the join target of the thread until that time. Each join frame has a link to the one-level outer join frame to keep the nested structures of join blocks. A weighted reference to the outer join frame is used as a link, and when a thread enters a new join block, its weighted reference is saved in the inner join frame (Figure 4.3 (d)). When exiting the join block (after synchronization), the saved weighted reference is restored.

Next, Figure 4.4 shows the (C macro) code for synchronization at the end of a join block. At lines 4–5, the thread locks the join frame and subtracts its weight from the counter, and then checks whether the counter value is 0 at line 6. If it is not 0, the thread must be suspended until all threads synchronize to it. A rendezvous is prepared in each join frame as a field `parent_fr` and the list of frames for the suspended thread is set into the field (line 10). A detailed method of suspending a thread is described in the next section.

4.3 Management of Method Frames

An OPA thread may invoke methods in a nested manner during its execution. In the most sequential language such as C, a series of sequential function calls are realized using a single stack. In C's runtime model, a processor executes a function using a stack frame allocated at the top of the stack. This means that the model does not assume multiple threads, so we cannot write C code that allocates multiple threads on the stack and executes them concurrently by context switch among them. For this reason,

```
1 #define WAIT_FOR_ZERO() do {                          \\
2   int a;                                              \\
3   join_frame *jf = pr->jf;                            \\
4   OPA_SPIN_LOCK(jf->join_lock);                       \\
5   a = jf->w - pr->jw;                                 \\
6   if (a) {  /* child thread exist yet */              \\
7     jf->w = a;                                        \\
8     {                                                 \\
9       f_frame *fr = ALLOC_FR(pr);                     \\
10      jf->parent_fr = fr;                             \\
11      OPA_RELEASE_LOCK(jf->join_lock);                \\
12      // save continuation                            \\
13      pr->callee_fr = fr; return SUSPEND;             \\
14    }                                                 \\
15  } else {   /* I am last thread */                   \\
16    OPA_RELEASE_LOCK(jf->join_lock);                  \\
17  }                                                   \\
18  FREE_FR(jf, pr->fv);                                \\
19 } while(0)
```

Figure 4.4: The code for `join` synchronization.

multithread implementations generally use a separate stack for each thread in the most simple manner, or they use frames allocated in heap instead of stack frames.

The OPA system uses heap frames for multiple threads. A method invocation is implemented as a sequential function call from a processor's scheduler (it is part of the runtime system written in C), and, at the beginning of the called C function's body, the content of the method's heap frame is moved onto the stack. (To avoid confusion, we refer to invocations in OPA as method invocations and C calls as function calls.)

Because heap frames must keep the caller-callee relations (implicit on stack), a callee's frame should point to the caller frame, and so one thread is represented as a list of frames in heap. To execute this form of a thread, the kernel code of the scheduler is implemented as follows:

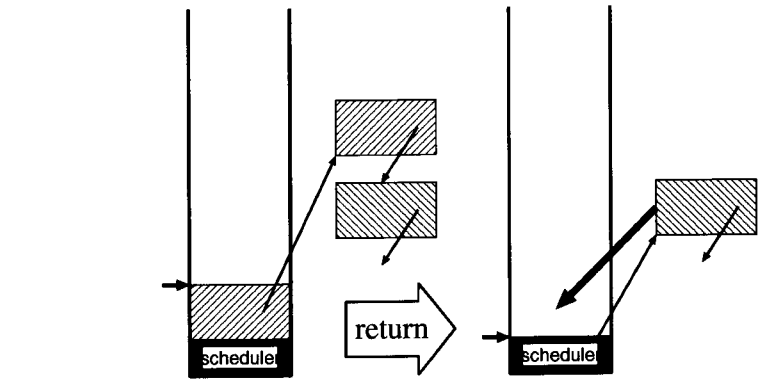
```
while(fr) { fr = (fr->f)(pr, fr); }
```

`fr` refers to a heap frame and a field `f` points to a function that starts (or resumes) the method execution. The OPA compiler generates two versions of C code (function) for each method, and here, the slow version code is pointed to from `f`. The fast version code is described later. `pr` passed to `f` as a first argument is a pointer to the data structure that keeps a processor's specific data (mentioned briefly in Section 4.1). `fr` is also passed as an argument so that slow version code can save/restore the state.

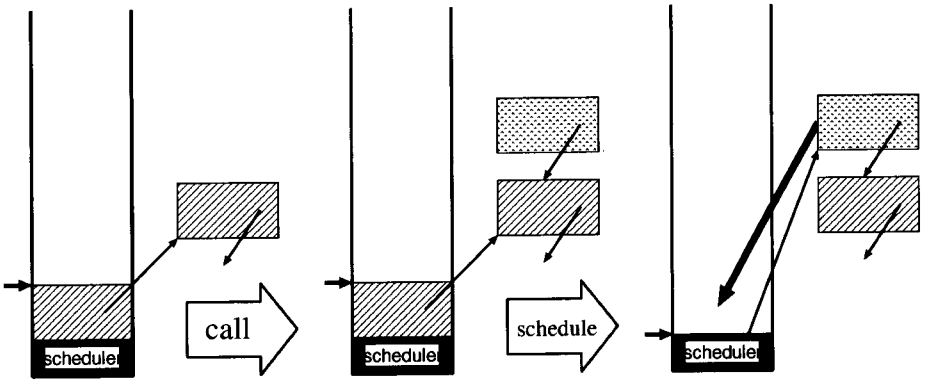
The slow version function returns a pointer to the heap frame that should be executed next. In this way,

- When a function completes its execution, as in Figure 4.5 (a), it returns a pointer to the parent's heap frame (the next element of the list). The method's return value is set into a fixed place (`pr->ret`) and the parent takes it out when it resumes. `ret` is defined as an union type so that it can hold any type of value (e.g., `ret.i` for `int`).

- When a function invokes a new method, as in Figure 4.5 (b), it allocates a heap frame for the new thread, initializes it (e.g., puts the method arguments in it), links a pointer from the child's heap frame to the parent's heap frame, saves the parent's continuation in its heap frame and returns a pointer to the child's heap frame to the scheduler.



(a)



(b)

Figure 4.5: Method invocation with the slow version C code.

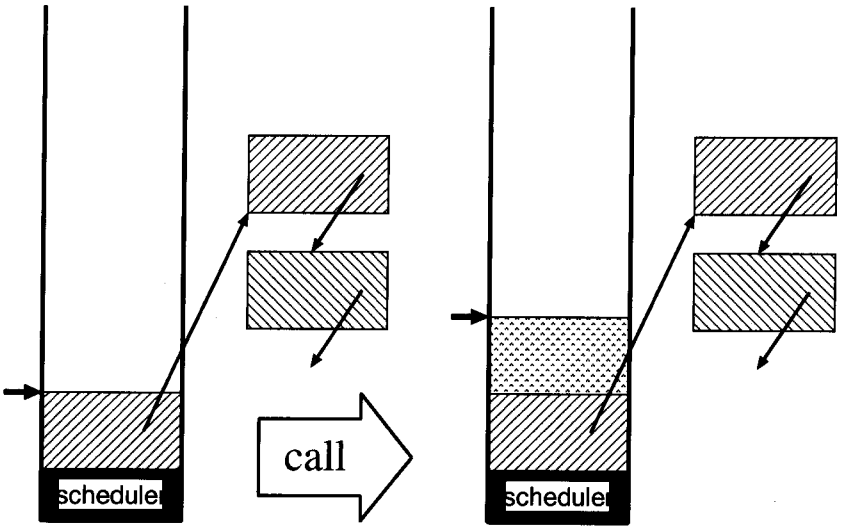


Figure 4.6: Method invocation with the fast version C code.

- When a function is suspended, it saves its continuation in its heap frame and returns 0 to the scheduler. 0 is also returned when a thread completes its execution (i.e., it has no more heap frame to be executed). If the scheduler gets the return value 0 in `fr`, it breaks the while loop to look for a next thread to execute.

In the above method, the states (continuations) of all methods are saved into heap frames and a stack frame is used only as a temporary place for caching the data needed by the currently running method. For more practical use of C stack, the OPA system uses almost the same technique for frame allocation as Hybrid Execution Model[21]. They use two kinds of frames (stack frames and heap frames) and two versions of C code (fast version and slow version).

- a method invocation is performed not by the scheduler but by the function currently running on the top of the C stack using a sequential call of the fast version C function (Figure 4.6). The parent's continuation is automatically saved in the stack. Also, explicit initialization of the child frame is not necessary since the initialization code is automatically generated by the C compiler.

- When a function completes its execution, it simply returns to the caller. (The problem of the method's return value is discussed later.)
- If the current thread is suspended, all stack frames must be evacuated into heap. For this, the fast version code allocates a new heap frame, saves its own state in it, and passes a pointer to the heap frame to the caller so that a list of heap frames can be constructed.

In Hybrid Execution Model, when a callee is suspended, a pointer to the callee's heap frame is returned as a return value of a C function call. If a callee completes its execution normally, it returns 0. The problem is that, besides a pointer to the heap frame, the callee must return the method's return value. In Hybrid Execution Model, the method's return value is stored in another place (in memory) allocated by the caller in advance. However, this method incurs memory access overhead to get the return value for each call. To reduce this overhead, the OPA system returns the method's return value as a return value of the C function call, and the special value `SUSPEND` (that is selected from rarely used value, e.g., `-5`) indicates that the callee may have been suspended. In such a case, the caller checks further if a pointer to the callee's heap frame is stored in a fixed place (`pr->caller_fr`). If it is not there, it can decide that the method's return value happens to be the same as `SUSPEND`.

As described above, a fast version function does not include the code for restoring the state saved in the heap frame, and a method invocation is realized as a sequential C function call (and a lightweight suspension check after the call). Thus, as compared to the method which uses only slow versions of code, the system performance is improved considerably.

4.4 Thread Creation and Scheduling

In this section, we present techniques for thread management used in the previous OPA implementation. Particularly, we explain thread creation, scheduling of runnable threads and dynamic load balancing. These techniques have a large influence on the whole

performance of the system. So, the inefficient techniques explained in this section are improved in Chapter 5.

First, we give a concrete shape (expression) to each thread in the OPA runtime system. It manages each thread with a meta *thread object* (that is invisible to programmer). The type of thread objects is defined as follows:

```
typedef struct thread_t {
  j_frame *jf;   /* a pointer to join frame */
  int jw;        /* weight  */
  f_frame *fr; /* continuation */
  thread_t *next;
}
```

The pair of `jf` and `jw` is the weighted reference and `fr` is the thread's continuation, that is, a pointer to (the front of) the list of heap frames. `next` is used to construct a waiting thread queue when the thread is suspended for mutual access to a certain object. In addition, the OPA system uses thread objects for thread identification; for example, it enables Java's `synchronized` methods which allows a thread holding a lock to acquire the same lock more than one times.

We must explain about a thread's continuation in more detail. Like future and touch operations of section 2.1.2, in order to pass a child thread's return value to its parent, a placeholder to store it must be allocated. (On the other hand, a method's return value is returned through `pr->ret` in the slow version code, since the parent method is always called from the scheduler immediately.) Also, the continuation must include the synchronization process (that is, subtraction of weights from the counter and (possibly) resumption of the waiting thread). For these purposes, an additional heap frame (`join_to`) is appended to the list of heap frames to process `fjoin` synchronization after completion of the thread. The heap frame has a field used as a placeholder (`ret`). A parent thread has a pointer to the heap frame and gets the return value from there after synchronization.

As mentioned earlier, if the scheduler gets the return value 0 in `fr`, it breaks the while loop to look for a next thread to execute. Each processor has a *ready thread queue*, a local pool for runnable threads, and the scheduler selects the next thread out of the

threads in the queue. When a thread that has been suspended becomes runnable, it is enqueued into the queue.

A newly created thread is also enqueued into the ready thread queue. Then, a thread creation is performed as follows:

1. creates a thread object,
2. splits weight between the parent thread and the child thread,
3. saves a continuation for starting from the beginning of the forked method's body,
4. appends the frame `join_to`, and
5. enqueues the thread object to the processor's local runnable thread queue.

The ready thread queue is also used as a communication buffer for thread migration, that is, a new thread object can be enqueued to another processor's ready thread queue. (This might be profitable in distributed-memory environments where data locality is also the key factor of the system performance.)

A ready thread queue is a doubly-ended queue (deque) and thread objects are enqueued at its tail. When a processor's stack becomes empty, it takes a thread from the local ready thread queue's tail and executes it. In other words, a processor uses its runnable thread queue like a LIFO queue. This is because it is more efficient to schedule a child thread first rather than its parent thread which must wait for the completion of the child thread.

When a processor becomes idle for its local ready thread queue becomes empty, it may steal a thread from the head of another processor's ready thread queue, thus enabling dynamic load balancing. The reason why the thread at the queue's head is chosen to be stolen is that it is expected to have the largest amount of work among all threads the processor owns. (More precisely, because of OPA's irregular parallelism (cooperative parallelism and mutual and exclusive parallelism), it is not always true that the thread at the queue's head is the largest.)

4.5 Sample Code: fib

To conclude this chapter, fast version C code for `fib` of Figure 3.8 generated by the OPA compiler is shown in Figure 4.7. (In the code, a comment that begins with `//` indicates that certain detailed operations are abbreviated for readability.) Each OPA's method is compiled to a C function. In addition to the method's parameters, the C function has another parameter `pr` that is a pointer to the data structure that keeps a processor's specific data.

In Figure 4.7, when a thread enters a join block, it allocates a new join frame `njf` (line 7). The pair of `pr->jf` and `pr->jw` holds the weighted reference that the current thread has at the moment. So, at line 8, the current thread stores a pointer to the outer join frame into the new (inner) one and, at line 9, the thread comes to refer to the inner one.

A thread creation is processed at lines 11–16:

1. creates a new thread object `nt` (line 11),
2. splits a weight for the newly created thread (line 12)
3. saves a continuation for starting from the beginning of the forked method's body (lines 13–14).
4. appends the frame `join_to` to process join synchronization after completion of the new thread (line 15).
5. enqueues the thread object to the processor's local ready thread queue (line 16).

Lines 18–24 correspond to a sequential method invocation. After returning from a sequential call of the corresponding C function, it is checked if the callee has been suspended (line 19). In such a case, since the caller belongs to the same thread as the callee, the caller also saves its own continuation (lines 20–21), links the callee to itself (line 22), and informs its own caller that it has been suspended (line 23).

When it exits from the join block (lines 25–26), it synchronizes using `pr->jf` as described in Figure 4.4. At line 27, it gets a return value from the placeholder after synchronization has completed.

```
1 int f__fib(private_env *pr, int n) {
2   f_frame *callee_fr; thread_t *nt;
3   f_frame *nfr; int x, y; f_frame *x_pms;
4   if(n < 2) return n;
5   else {
6     /* enter a join block */
7     frame *njf = ALLOC_JF(pr);
8     njf->bjf = pr->jf; njf->bjw = pr->jw;
9     pr->jf = njf; pr->jw = njf->w;
10    /* create a new thread */
11    nt = ALLOC_OBJ(pr, sizeof(thread_t));
12    nt->jf = pr->jf; nt->jw = SPLIT_JW(pr, pr->jw);
13    nt->cont = nfr = MAKE_CONT(pr, c__fib);
14    // save continuation (including n-1)
15    x_pms = MAKE_CONT(pr, join_to); nfr->caller_fr = x_pms;
16    enqueue(pr, nt);
17    /* sequential call */
18    y = f__fib(pr, n-2);
19    if((y==SUSPEND) && (callee_fr=pr->callee_fr)) {
20      f_frame *fr = MAKE_CONT(pr, c__fib);
21      // save continuation
22      callee_fr->caller_fr = fr;
23      pr->callee_fr = fr; return SUSPEND;
24    }
25    WAIT_FOR_ZERO();
26    pr->jf = pr->jf->bjf; pr->jw = pr->jf->bjw;
27    x = x_pms->ret.i;
28    return x+y;
29  }
30 }
```

Figure 4.7: Compiled (pseudo) C code for `fib`.

In summary, our previous implementation schemes of OPA incur unacceptable overhead for fine-grained multithreaded programs.

Chapter 5

Optimizations for Parallel Processing

In this chapter, we improve the implementation schemes for fork-join constructs described in the previous chapter[27, 28]. A goal of the improvement is to reduce fork-join constructs' overhead close to zero so that the absolute execution time of fork-join style parallel programs on a single processor becomes close to the sequential version of programs, while it supports good load balancing.

5.1 Basic Idea

Implementations of OPA's advanced features have a large impact on the performance of the fork-join constructs. For example, to support synchronized method, our OPA implementation allocates a new thread object at each thread creation. To support cooperative parallelism and exclusive parallelism, it employs thread suspension/resumption mechanism and uses dynamically created data structures for synchronization, i.e., join frames. Also, if join target was not determined with dynamic scope, instead of using join frames, counters for `join` synchronization might be embedded in thread objects or thread continuations.

In OPA, those data structures are indispensable and the percentage of their runtime overhead is large in each thread creation and join synchronization. In this chapter, we propose three techniques which reduce these overhead. Their common key concept is

laziness. Laziness means that we delay certain operations until their results become truly necessary.

First, allocating heap frames can be lazily performed. In Section 4.3, we allocated a stack frame at first for efficiency, and it remains there until it prevents other thread's execution. In this chapter, we delay heap frame allocation not only for each sequential method invocation, but also for each thread creation; allocating heap frames for both the parent thread and the child thread can be delayed until another processor becomes idle and steals work from it.

Second, a *task* can be lazily created. In this thesis, a task is (a data structure of) a schedulable active entity that does not correspond to a single (language-level) thread. A blocked thread becomes a task to release the processor and the stack it uses. Also, for the purpose of load balancing, each thread may become a task to move around the processors. Our approach decreases the number of actually created tasks while it keeps good load balancing. Furthermore, we delay some other operations related to thread object creation and, as a result, we can make the cost of thread creation close to zero. Third, data structures for synchronization (join frames) can be lazily made.

The global organization of the OPA runtime system described in Section 4.1 is extended as follows (Figure 5.1). Two data structures are added:

lazy task queue (LTQ) it includes pointers to thread objects and keeps track of the threads currently running on the C stack.

join stack it includes pointers to join frames and used by threads currently running on the C stack for finding its join target; they cannot directly refer to its join frame since it might not yet be allocated.

Also, some fields are added in the processor specific data area (`pr`) used for interprocessor communications of work steal.

5.2 Lazy Task Creation

As described in Section 4.4, the conventional OPA system creates a task, i.e., a full-fledged thread object, at the time of thread creation. In other words, every thread

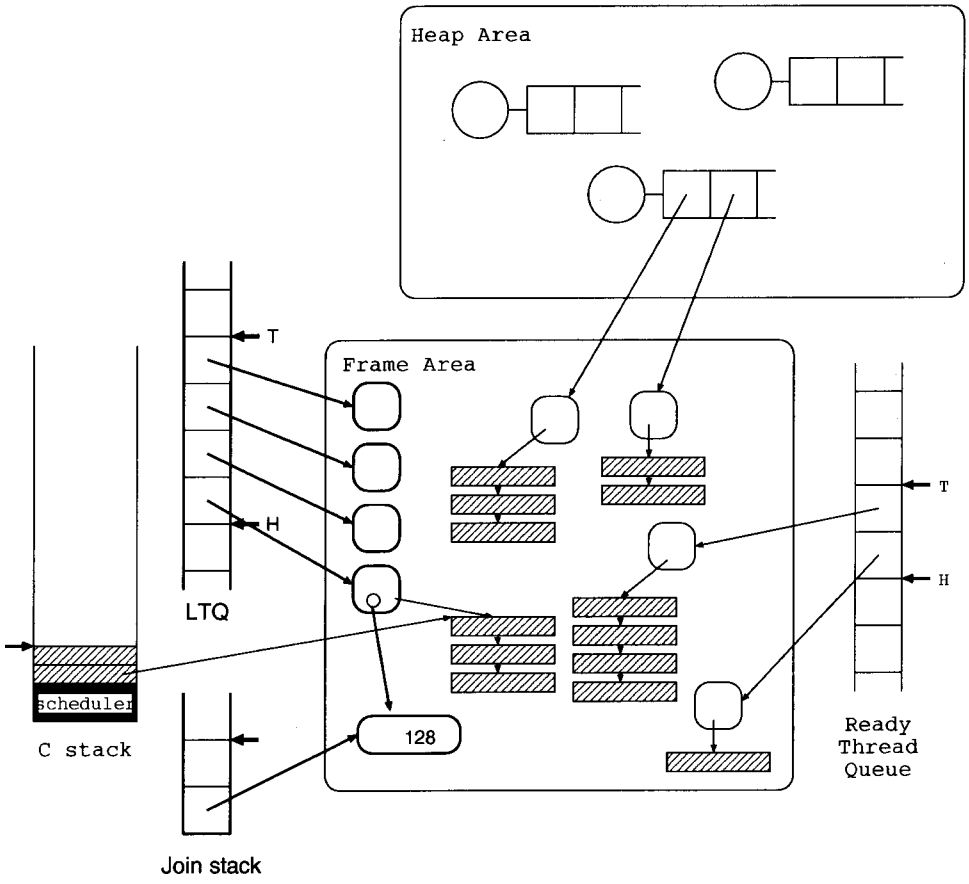


Figure 5.1: Organization of the extended OPA runtime environment.

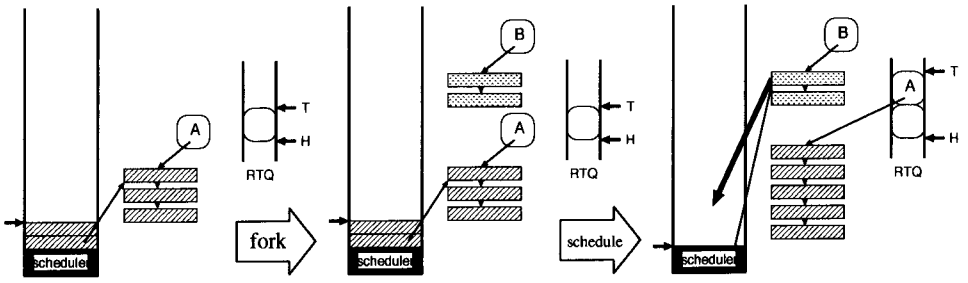


Figure 5.2: Thread creation with child-first scheduling policy.

creation is realized as "eager" task creation.

Lazy Task Creation (LTC)[20] is used for an efficient Multilisp[12] implementation originally. LTC dynamically creates tasks only when thread migration must be performed for load balancing thus relieving the programmer from having to think about thread granularity. In brief, the basic idea of LTC is as follows:

- At thread creation, the processor starts a child thread's execution prior to the parent. The parent's continuation is (automatically) saved in stack.
- An idle processor (thief) steals a task by extracting (a continuation of) a thread from the bottom of the processor's stack and making a task from it.

The advantages of LTC are (a) saving the parent's continuation takes no more cost than a sequential function call. (b) It enables dynamic load balancing by allowing a thief to extract a thread from the victim's stack.

Now, consider that we may realize these two processes in C. First, saving a parent's continuation is accomplished by a normal C function call, so we need no more consideration. Second, we encounter a problem that we cannot extract a continuation from a C stack in a portable manner.

In contrast, the Multilisp system manipulates the stack at assembly level. When it creates a thread, a pointer to the parent's stack frame is saved in the processor's local queue (Multilisp's LTQ). A thief can find victim's bottom continuation using LTQ.

To realize this process at C level, a parent's continuation must be saved into its heap frame in advance. The creation of a new thread is performed as shown in Figure 5.2.

1. creates a full-fledged thread object: that is, creates a thread object, splits weight, saves a continuation (including `join_to`),
2. suspends the parent thread and enqueues it to runnable thread queue, and
3. returns a pointer to the child thread's continuation to the scheduler.

As compared with the previous OPA's method, the allocation of heap frames is *not* delayed. It only changes the scheduling policy so that the child thread is executed first on the processor. However, preparing the full-fledged parent seems to be necessary for work steal.

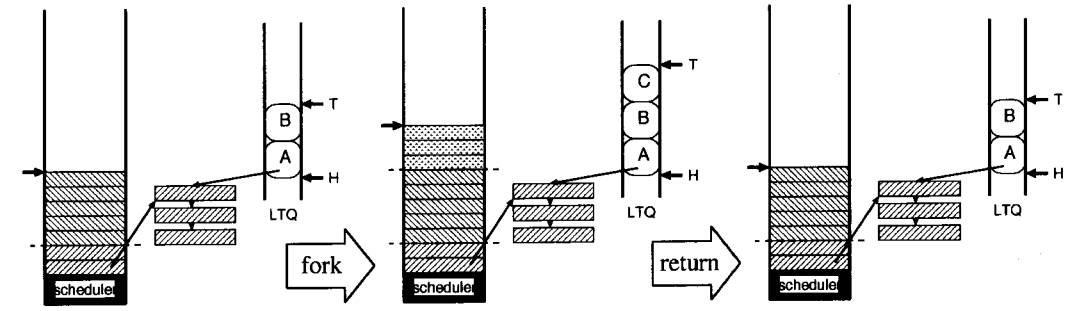
This method is similar to what is employed in the Cilk implementation[7]. Cilk also saves a parent's continuation into its heap frame in advance. But, Cilk forks a new thread by calling the fast version function directly from the caller as in original LTC. This is possible in Cilk because it always allocates a heap frame and saves a continuation in it for every cilk function. This means that Cilk's continuations are always stealable. On the other hand, creating a new thread as a direct function call from the caller is impossible in OPA because it delays allocations of heap frames for method invocations, that is, continuations are saved only in the C stack. So, at the time of a thread creation, the caller thread must be explicitly suspended for evacuating its own stack frames.

In this method, a thief can steal a thread without manipulating victim's C stack although it incurs a certain amount of overhead on every thread creation, particularly for saving a parent's continuation. Instead, we want to keep the parent's continuation on the C stack at the time of thread creation. Thus, heap frame allocation is delayed until a thread really needs to be suspended (or be stolen). In order to solve this problem, we adopt a message-passing implementation of LTC[4]. Its feature is that a thief does not access another processor's local data including its stack. The thief simply sends a message for a task request to a randomly selected victim and waits for the response. A victim, when it notices the request, extracts a continuation from its own stack, makes a task from it, and sends it back to the thief. We use polling to check a request message,

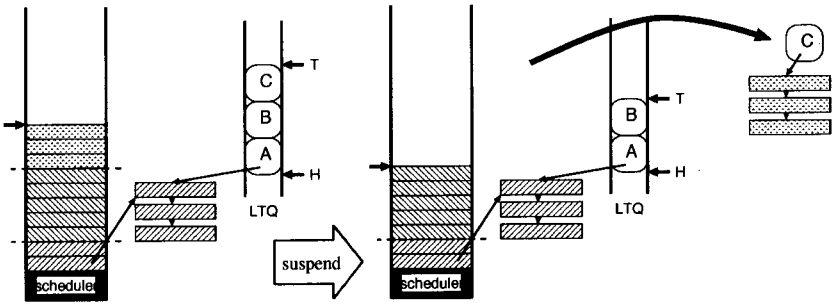
and an efficient polling technique is found in [6]. Polling is also used for exception handling and garbage collection in the OPA implementation.

Figure 5.3 describes the behavior of the message passing implementation of LTC in OPA. Thread objects which correspond to the running threads on the C stack is kept in LTQ (not ready thread queue) in the same order as the stack. If heap frame allocation for one of those threads is delayed, its thread object is not full-fledged. At thread creation in Figure 5.3 (a), the parent's thread object is enqueued at the tail of LTQ. Then, a new thread object for the child is created and the fast version C function is called directly from the parent. After the normal return from the child thread, the parent thread takes out its own thread object out from the tail of LTQ and continues its execution. When the current thread (at the top of the C stack) is blocked (Figure 5.3 (b)), only this thread is suspended and evacuated to heap as before. The difference appears when the control returns to its parent thread (not the scheduler). The parent thread can notice the suspension of the child in the same way as suspension check after a method invocation (i.e., comparing the return value with `SUSPEND`). The parent thread takes out its own thread object out from the tail of LTQ and continues its execution. When a processor receives a request message (Figure 5.3 (c)), in order to extract a continuation from the bottom of the C stack, it temporarily (internally) suspends all continuations (threads) above it. The way how each thread is suspended is the same as (b). Each thread decides whether it must be suspended or it can continue (i.e., steal or just a suspension of its child thread) by examining the processor's message box only for steal messages (`pr->thief_req`).

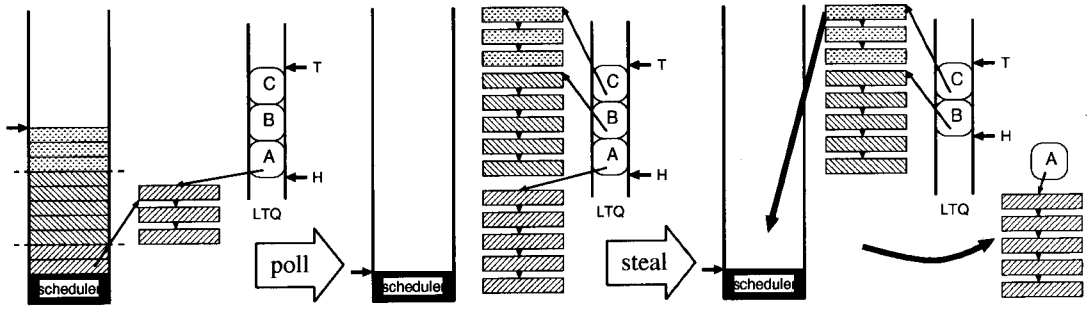
After these operations, the bottom continuation has been converted into a task at the head of LTQ and can be sent to the thief. The disadvantage is that it converts all the continuations on stack into tasks, not only the bottom continuation. However, LTC assumes well-balanced divide-and-conquer programs, and during the execution of such programs, we expect for only a few times of stealing to happen. Also, even in unbalanced programs, we avoid the overhead as follows: the victim does not resume all tasks in the task queue with stack frames (i.e., completely recover the C stack) to restart, but the victim resumes a task taken from its tail with slow version code, and uses the remaining tasks for the later request without the conversion overhead.



(a) fork



(b) suspension



(c) steal

Figure 5.3: Message passing implementation of LTC.

From the above discussions, the order of means by which a processor (scheduler) finds a next thread to execute: (1) takes from the tail of LTQ, (2) takes from the head of the ready thread queue, and (3) sends a request message for steal.

So far, we present how we can extract the bottom continuation from C stack. However, for now, only two of five operations of task creation (listed in the Section 4.4), saving continuation and queuing, have we done lazily. In order to bring the cost of thread creation as close to that of sequential call as possible, we need adequate modifications to the remaining three operations.

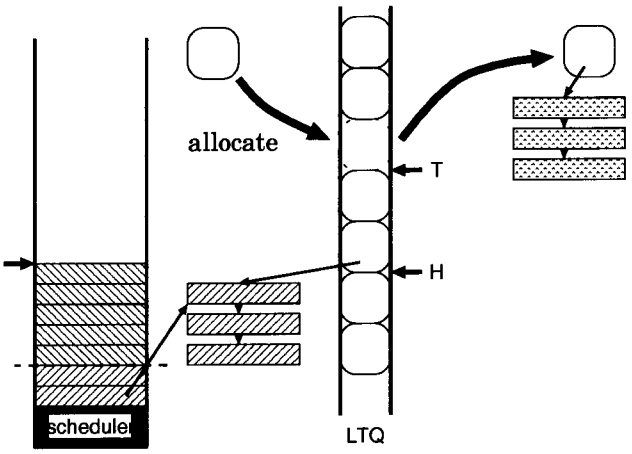
The important point is that a child thread is called from a parent as normal function call. If a child can continue its execution until the end without blocking (i.e., on C stack):

- the thread's return value can be passed through the function's return value (not through placeholder), and
- since the child thread always complete its execution before the parent thread reaches the end of join block, there is no need to split weight between them.

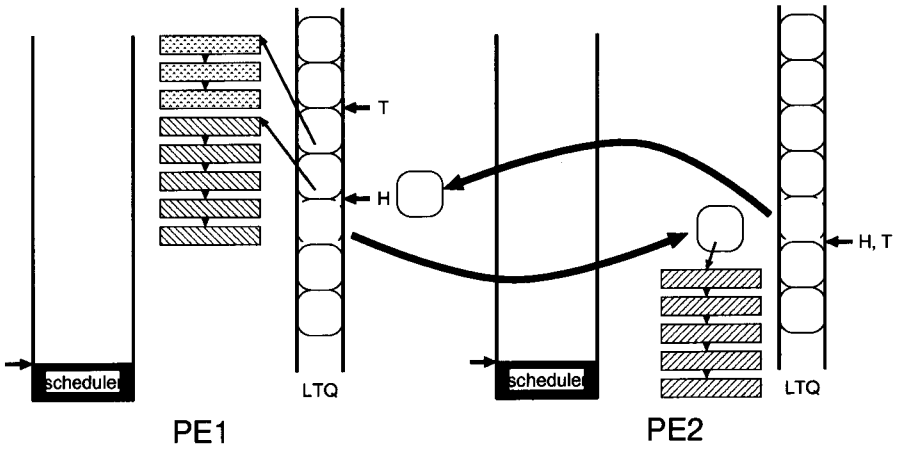
Only when a child thread has blocked, a `join_to` frame for synchronization (and a placeholder) is appended.

In this way, we can notice that, when neither steal nor suspension happen, we do not use the contents of thread objects which is created at each thread creation at all (at least, in current implementation): heap frames (continuations) are not allocated and weighted references are not configured. Thus they are only used for thread ID.

It means that a thread object needs to be unique only while the corresponding thread lives in stack. Then, it is redundant to allocate a thread object at each thread creation in the same place of LTQ for many times. We decide not to free the thread object at thread's termination, and reuse it for the next thread creation by keeping it in LTQ. Thread objects are allocated at initialization of LTQ. When a new thread is created, the tail pointer of LTQ indicating the current thread is incremented. Stealable thread objects (tasks) are now between the head and the tail (excluding what the tail points to). When a thread leaves stack for suspension, since it brings its thread object together, a substitute one is allocated as in Figure 5.4 (a). When a processor resumes a task, it



(a) suspension



(b) steal

Figure 5.4: Thread object management in LTQ.

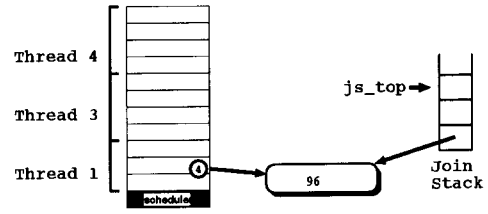
frees the thread object that has been already in task queue's head. Altogether, when a thief steals a task from a victim, the two thread objects at the head of their LTQs are exchanged as in Figure 5.4 (b).

5.3 Laziness for Join Frame Management

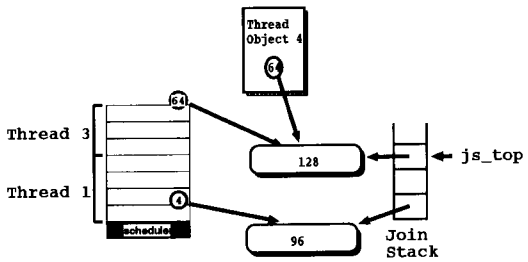
In the previous section, there is a lazy normalization scheme in which we do not have to split weight at all times. In this section, we extend this laziness to the process of entering/exiting a join block. As described before, it can be said that there is an implicit synchronization between a parent and a child thread on C stack. Then, as long as all the threads that synchronize to a certain join block execute in the same stack, that is, no heap frame is allocated for those threads, there is no need to use a counter for this join block. In such a case, the corresponding join frame exists only for maintain the depth of nested join blocks (recall that they have a pointer field to the outer join frame).

Alternatively, instead of allocating join frames, we prepare a stack (`join stack`) for each processor. It stores pointers to some (not all) join frames. We can delay the allocation of a join frame until heap frames are allocated for some threads that synchronize at the join point. In that case, the corresponding element of the join stack is empty. The reason why we use a stack, not a simple depth counter, is that we want to allocate a join frame and store it into the join stack when the above condition is not satisfied. By probing the element of the join stack whose depth is equal to the depth of nested join blocks, threads running on the C stack can find whether the join point has been allocated a join frame.

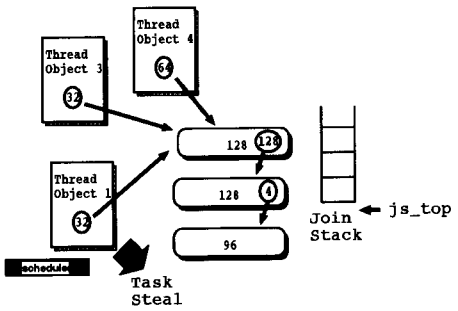
The fork-join data structure in Figure 4.3 is improved to Figure 5.5. At initial state, the join stack is empty. After entering join blocks for several times, it only needs to increment the top pointer (Figure 5.6 line 7) and the state becomes to Figure 5.5 (a) (Be sure that there is one join frame at the bottom, and this is what resumed task has already had.) In the case of exiting a join block without making the join frame, it only needs to decrement the top pointer. If certain thread blocks and join stack's top has no pointer to a join frame, it allocates a new join frame, set the pointer to stack top, and split the weight (Figure 5.5 (b).) When extracting a continuation for steal, all the



(a) No join frame



(b) A single join frame is allocated



(c) All join frames are allocated

Figure 5.5: Lazy allocation of join frames.

join frames are allocated and split weights for all the threads. In addition, functions that executed in its local join block need to link a pointer between join frames to remain nested join block structure in heap. The final state of join block structure looks like Figure 5.5 (c).

5.4 Sample Code: fib

To conclude this chapter, we describe the details with Figure 5.6. It is fast version C code for `fib` of Figure 3.8 with laziness and a thread is created in lines 13–26. A function call at line 14 corresponds to a child thread's execution. The important point is that a child thread is called from a parent as normal function call. If a child can continue its execution until the end without blocking (i.e., on stack),

- the thread's return value can be passed through the function's return value (not through placeholder), and
- since the child thread always complete its execution before the parent thread reaches the end of join block, there is no need to split weight between them.

In Figure 5.6, at line 14, thread's return value is set to `x` directly.

A suspension check at line 15 is the same as the conventional one. That is, while extracting the bottom continuation, all functions on stack act as if all threads were blocked. Only when a child thread has blocked, a frame for synchronization (and a placeholder) is appended (line 16), and only in such a case the parent thread examines a pointer to the placeholder and get a return value (line 39). (Note that weight has been split at the point of blocking (e.g., after polling) and set into the child's thread object.)

At line 17, it checks a task request. Since the non-zero value (the thief processor ID) means that this processor is requested to extract the bottom continuation, it starts suspending the parent thread (lines 18–23). `ltq_ptr` points to the corresponding thread object in the task queue. The way of splitting the weight differs from that of conventional code, and we explain this in the next subsection.

Ultimately, in the case of no suspension or steal, code for a thread creation has reduced to lines 13–15, and 26. In other words, the overhead as compared with sequential call is

```
 1 int f__fib(private_env *pr, int n) {
 2   f_frame *callee_fr; thread_t **ltq_ptr = pr->ltq_tail;
 3   int x, y; f_frame *x_pms = NULL;
 4   if(n < 2) return n;
 5   else {
 6     /* enter a join block */
 7     pr->js_top++;
 8     /* polling here */
 9     if(pr->thief_req) {
10       // suspension code here
11     }
12     /* create a new thread */
13     pr->ltq_tail = ltq_ptr+1;
14     x = f__fib(pr, n-1);
15     if((x==SUSPEND) && (callee_fr=pr->callee_fr)) {
16       f_frame *x_pms = MAKE_CONT(pr, join_to); callee_fr->caller_fr = x_pms;
17       if(pr->thief_req) {
18         f_frame *fr = MAKE_CONT(pr, c__fib);
19         (*ltq_ptr)->jf = *(pr->js_top); (*ltq_ptr)->jw = SPLIT_JW(pr, pr->jw);
20         (*ltq_ptr)->cont = fr;
21         // save continuation
22         SUSPEND_IN_JOIN_BLOCK(pr);
23         pr->callee_fr = fr; return SUSPEND;
24       }
25     }
26     pr->ltq_tail = ltq_ptr;
27     /* sequential call */
28     y = f__fib(pr, n-2);
29     if((y==SUSPEND) && (callee_fr=pr->callee_fr)) {
30       f_frame *fr = MAKE_CONT(pr, c__fib);
31       // save continuation
32       callee_fr->caller_fr = fr;
33       SUSPEND_IN_JOIN_BLOCK(pr);
34       pr->callee_fr = fr; return SUSPEND;
35     }
36     /* exit join block */
37     if(*(pr->js_top)) {
38       WAIT_FOR_ZERO();
39       if(x_pms) x = x_pms->ret.i;
40     }
41     pr->js_top--;
42     return x+y;
43   }
44 }
```

Figure 5.6: Compiled (pseudo) C code for `fib` with laziness.

only a pair of increment/decrement of the task queue pointer and a check for suspension.

Chapter 6

Extending LTC for Iterative Computation

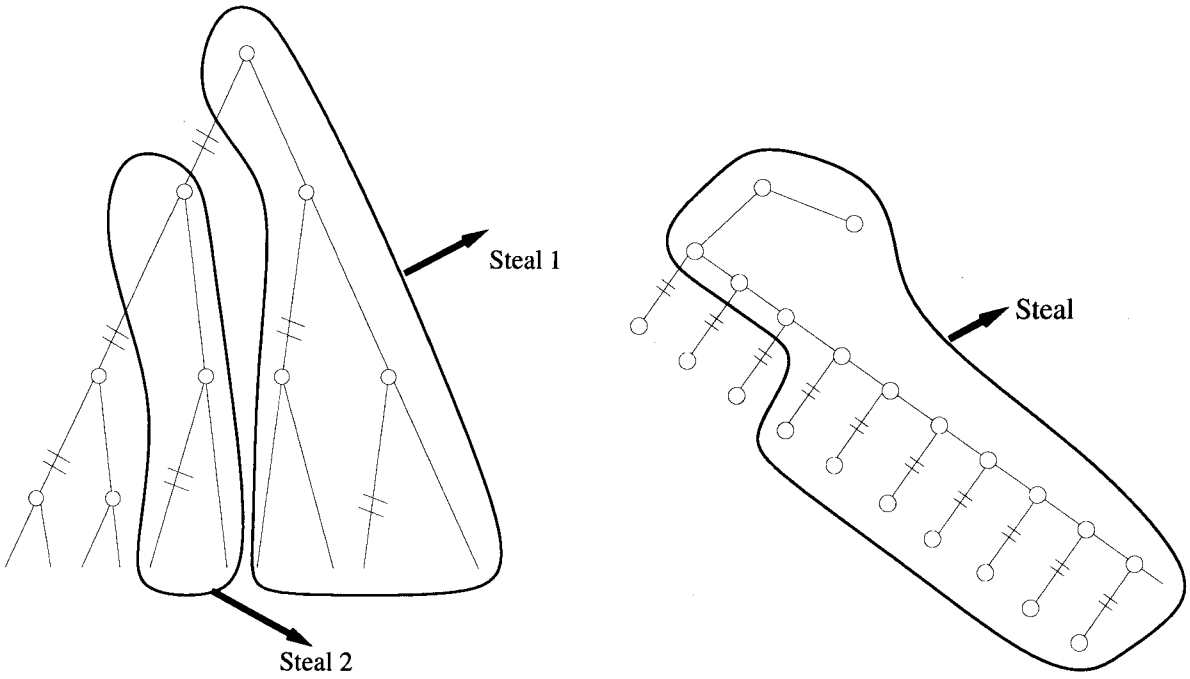
In this chapter, we address the problem that LTC cannot achieve good performance for programs that fork many threads in a iteratively executed loop construct, such as `for` or `while`.

6.1 Inefficiency of par Call in a Loop

Essentially, load balancing by LTC is effective only in tree-recursive parallel programs. The following code is an example of such code:

```
f(...) {
  if(...){ ... }
  else {
    par f(...);
    f(...);
  }
}
```

This code assumes that there is no significant difference in the amount of work that a newly created thread and the parent continuation have, that is, we can say that a thief can get the moderate work (the parent continuation) from the viewpoint of load balancing (as illustrated in Figure6.1 (a)). In Figure 6.1, each node represents a single



(a) Tree recursive (b) Loop

Figure 6.1: The amount of work between thief and victim.

method invocation or a single iteration of a loop. A dotted branch means a thread creation (`par` call).

On the other hand, the OPA language, as well as Java or C, has loop constructs and it enables to create new threads iteratively. The following is an example of such code:

```
for(int i = 0; i < N; i++)
  par g(i);
```

The problem of executing this code (we call it `for-par` loop) in LTC is that the work of the parent continuation would be much larger than the work of the child thread as in Figure 6.1 (b). Assuming that all iterations have almost the same amount of work, the continuation of a `par` call at the iteration $i = 0$ includes the remaining $N - 1$ iterations, so the ratio of the amount of work is about $N - 1 : 1$.

When the amounts of work for a victim and a thief are unbalanced like above, the one that is assigned the smaller work would immediately becomes idle and causes a work steal. In the above case, the number of work steals is equal to the number of iterations, thus incurs significant overhead.

In the next section, we propose several efficient load-balancing methods those are also effective in OPA's loop constructs. The above problem also arises when we write tail-recursive programs on Scheme implementations which originally use LTC for load balancing. Our methods can also be applicative to those cases in principle.

6.2 Extension of LTC

6.2.1 Dividing `forall` Style Loops

First, let us consider the following code:

```
for(int i = 0; i < N; i++)
  par g(i);
```

For this `for-par` loop, we can identify the loop control variable and the count of iterations at compile time. Also, the loop body contains only a `par` call and the values needed to the `par` call (a target object, a method to be invoked, and arguments) do not depend on the results of the previous iterations, that is, this `for-par` loop is a `forall`-style loop that has inter-iteration parallelism.

If a message for a task request arrives at a victim while it executes a `forall`-style `for-par` loop, it is desirable to leave the first half of the remaining iterations for the victim itself and sends back the second half (and the continuation of the loop) to its thief. (In figure 6.2, the size of A is the same as that of B.) If messages arrive at a victim from multiple idle processors, it is possible to divide the remaining iterations equally among the thieves and the victim.

The task steal operation of LTC described in Section 5.2 is modified as follows[25, 26]. When extracting a continuation from the head of LTQ, the victim increases the control variable `i` of the continuation by the count of iterations for the thief. Instead of the continuation, it puts a new task that only contains a `for-par` loop ranging over the

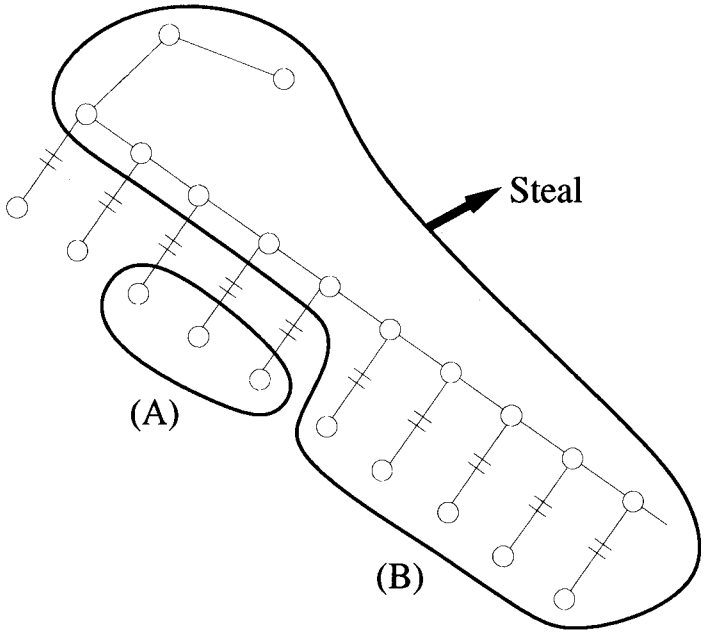


Figure 6.2: Task steal in `for-par` loop execution.

iterations for the victim at the head of LTQ. If another request message arrives again later, it splits the range of `i`, creates a new task that only contains the same `for-par` loop (it only differs in the range of `i`), and sends it back to the thief.

6.2.2 Stock Mode Execution

Generally, a `for-par` loop's body may contain any kind of statement(s). Consider the following code:

```
for(int i = 0; i < N; i++) {
  j = h(i, j);
  par g(j);
}
```

In this code, the argument `j` of the I_1-th `par` call to `g()` is decided only after performing the `for-par` loop for $i = 0$ to $I_1 - 1$, and the (sequential) calls to `h()` for $i = 0$ to I_1

must be completed before the I_1-th `par` call to `g()`.

Also, in the following code:

```
for(List xs = li;xs != null;xs = xs.cdr)
  par xs.car.f();
```

each `par` call can be performed only after finding the corresponding element of the list by executing the sequential part of the loop.

In general, if a `for-par` loop contains any statement(s) except `par` call, because of their side-effects, we cannot extract any `par` call from the loop prior to their execution. Additionally, since its control variable might be substituted (updated) irregularly, the number of the remaining iterations is not known exactly.

Therefore, in those cases, we split a `for-par` loop as follows:

- Since the number of the remaining iterations is not known, the number of the iterations for the victim (A in Figure 6.2) is determined heuristically: it is large enough to limit the number of steals and small enough to avoid cache overflow caused by the stored information for A.
- Before the victim extracts the continuation (B in Figure 6.2), it completes the execution of the sequential code for A.

To realize this, we prepared an alternative execution mode (namely, *stock mode*):

- Every statement except `par` call is performed normally.
- Instead of an actual call, each `par` call's information is stored (stocked) into a specific table (prepared for each processor).

For a task steal, before extracting the bottom continuation, we execute the continuation in the stock mode so that we can store `par` calls into the table in advance. Just after the table is filled with `par` calls, the processor is resumed to the normal mode and the continuation is sent back to the thief as B. If another request message arrives again later, the victim simply splits the `par` calls just in the middle of the table.

6.2.3 Performance Model

In this section, we model the execution of a non-`forall`-style `for-par` loop (that is, the number of its iterations cannot be analyzed at compiled time) and verify that it is more efficient to execute it in the stock mode than to execute in normal LTC.

In the following paragraphs, we analytically estimate the execution time of three cases: tree recursive code in normal LTC, `for-par` loop code in normal LTC, and `for-par` loop code in the stock mode.

Tree recursive code in normal LTC Assuming that threads are so fine-grained that the work can be divided equally among the processors, we can estimate the number of task transfers (that is nearly equal to the number of task creations) as follows. Let P be the number of processors, T_a the total amount of work (that is, the execution time on 1 PE), T_1 the amount of work of the minimum task, and C the time required for a task transfer, the number of task transfers is approximately $(P-1)\log_2((T_a/P)/(T_1+C))$.

For example, when P = 10PE, T_a = 10 seconds, T_1 = 0.001 seconds, and C = 0.001 seconds, the number of task transfer is about 80 times and the overhead of task transfer per processor is negligible (80 times * 2PE * 0.001 seconds / 10PE = 0.016 seconds).

`for-par` loop code in normal LTC For example, in the following code:

`for(`s_0`;`c`;`s_2`){`s_1`;par` p`;}`

throughout this loop (except at the beginning and the end), the pattern "after s_2, c, s_1 is sequentially executed, p is forked" is repeated.

First, we examine its execution in normal LTC on a single processor. Let T_q be the execution time of the sequential part (s_2, c, s_1) and T_p the concurrent part (`par` p), the execution time per iteration is $T_q + T_p$.

Next, we examine its execution in normal LTC on multiple processors. Let T_s be the time spent on stealing a task from a victim and T_g be the time spent on giving a task to a thief, the execution time per iteration is $T_s + T_q + T_p + T_g$.

From these examination, we can estimate that the execution time is increased in the ratio of $(T_s + T_g)/(T_q + T_p)$. Here, T_g contains the time for suspension/resumption and

the time for task creation/transfer. Furthermore, T_s contains time to wait for a thief to notice a message by polling, as well as wait for T_g. Because a thief may be forced to wait while another thief succeeds in stealing from the same victim, the increase of the number of processors cause the increase of waiting time, thus resulting in poor performance, that is, insufficient speedup results.

`for-par` loop code in the stock mode Let m be the number of iterations to be stored, T_o the time to store a single **`par`** call, and T_r the time to take a **`par`** call out of the table, we can estimate that the execution time during m iterations is $T_s + m(T_q + T_o) + T_g + m(T_r + T_p)$.

Here, T_s contains the time to wait for the completion of storing m calls ($m(T_q + T_o)$). When we assume that m is relatively large and so that the time to wait for a victim to notice a message by polling (included in T_s) divided m is negligible, we can calculate that the execution time per iteration is $2(T_q + T_o) + (T_r + T_p)$.

Therefore, we can conclude that the stock mode has an advantage over normal LTC if $T_s + T_g > T_q + 2T_o + T_r$ is satisfied. Here, the right side members of this inequality are likely to be relatively small since all these operations are performed within a single processor (that is, there is no need for synchronization/cooperation).

Chapter 7

Implementation of Exception Handling

In this chapter, we present efficient implementation techniques for exception handling on the OPA system which delays various operations as described in the previous chapter.

First, we explain how an exception that is thrown and handled within a single thread is implemented.

Next, we describe how an exception that is thrown during the parallel execution can be propagated as shown in Section 3.5. Moreover, we can implement them by efficient techniques that match the system's thread scheduling policy. (iHin our OPA system, it means LTC.)

7.1 Exception Handling within a Thread

To realize exception handling in OPA, we need to implement the following mechanisms:

`throw` statement: an exception that is thrown in a function is assigned to the function's local variable `ex` temporarily. As described later, an exception may be assigned to other places until it reaches a handler.

`catch` handler: the code for a handler is generated into the C code (function) for the method that includes the `try-catch` statement. A handler checks local variable `ex`.

```
1   /* sequential call */
2   x = f__f();
3   if((x==SUSPEND) && (callee_fr=pr->callee_fr)) {
4     if(ex = pr->ex) {
5       goto CONT_EX_0;
6     }
7     f_frame *fr = MAKE_CONT(pr, c__f);
8     // save continuation
9     callee_fr->caller_fr = fr;
10    pr->callee_fr = fr; return SUSPEND;
11  }
12  /* throw statement */
13  ex = ALLOC_OBJ(pr, sizeof(objbody_MyException));
14  goto  CONT_EX_0;
15 CONT_EX_0:
16  if(instance_of(ex, cls_MyException)) {
17    ...;
18  else {
19    pr->ex = ex;
20    pr->callee_fr = EXCEPTION; return SUSPEND;
21  }
```

Figure 7.1: The code for exception handling within a thread.

If it can handle the exception, the body of the `catch` clause is executed; otherwise, the exception is re-thrown from this point.

For example, the following OPA code is compiled to the C code shown in Figure 7.1:

```
try {
  f();
  throw new MyException();
} catch(MyException e) {
  ...;
}
```

Method local exception mechanisms are realized as follows. The beginning of a handler code is identified with an unique label in its C function (at line 15). When an exception is thrown in the same function, it is assigned to `ex` and then control jumps to the beginning of the inner most handler code by a `goto` statement (lines 13–14). The exception may be re-thrown for some times in a function since `try-catch` constructs can be nested in a method.

An exception that is not caught within a function (method) must be propagated to its caller. Therefore, the callee puts the exception into `pr->ex` and then return to the caller (lines 19–20). The caller checks `pr->ex` just after the call (lines 4–6) and, if an exception is set, it re-throws the exception by method local exception mechanisms (that is, it jumps to the inner most handler). The callee called with fast version code returns a value `SUSPEND` to the caller so that the caller have only to check the return value in most cases instead of checking `pr->ex` after every call, while the callee called with slow version code returns a pointer to the caller (as before) and the caller always checks `pr->ex`.

So far, we have not discussed `finally` clauses. In brief, the code for a `finally` clause is also generated into the C code for the method that includes the `finally` clause. Inside a `try` block that has a `finally` clause, a statement that escapes from the `try` block, such as normal completion of the `try` block, `return` statement, `break` statement, and throwing (or re-throwing) an exception, is followed by a `goto` statement which jumps to the beginning of the inner most finally block. After the execution of the `finally` block, the control is transferred according to the context of its escape.

7.2 Exception Handling during Parallel Execution

As described in Section 3.4, in the OPA language, if an exception cannot be handled by a thread, the exception is propagated to the join target of the thread, which then stops the other threads sharing the same join target. Also, in order to stop the other threads sharing the same join target, we defined that they automatically throw a special exception `stopped`.

To realize these features, we must implement the following:

- how to propagate an exception to a join frame.

- how to direct the other threads to throw an `stopped` exception.

7.2.1 Propagation of Exception

We added a field `ex` to each join frame so that an exception that is propagated from a certain thread can be stored into it. Each assignment to `ex` must be done in the same mutually exclusive manner as decreasing a counter of weighted reference counts. Note that we do not override the `ex` field if an exception has been already assigned to it, since we have decided that the survivor is the exception that reaches first in Section 3.4.

A thread that enters a join block checks `jf->ex` just after the `join` synchronization. If it finds an exception in `jf->ex`, it re-throws the exception immediately. The way of re-throwing an exception (`goto` statement) is the same as before, and the code is inserted between line 17 and line 18 of Figure 4.4.

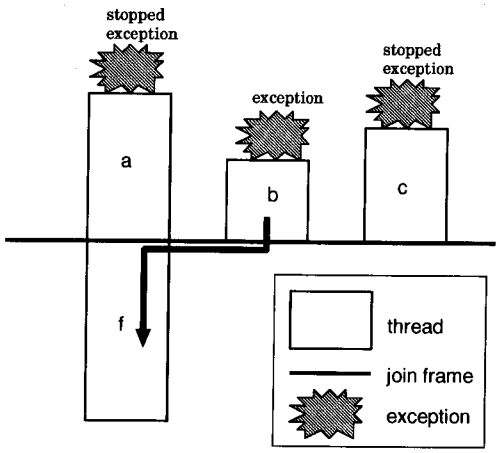
The way of propagating an exception to a join frame in fast version code differs from the way in slow version code since, in fast version code, allocating the join frame may be delayed using laziness.

In a sample program of Figure 7.2 (a), we can picture the configuration of the context as a cactus stack in Figure 7.2 (b), where an exception is thrown in a thread that executes `b()`. In the actual implementation, the exception is passed along the C stack as shown in Figure 7.2 (c). Also, a part of the compiled C code for Figure 7.2 (a) can be found in Figure 7.3. As in the case of calling a fast version function as a method invocation that is described in the previous section, a fast version function as a new thread returns a value `SUSPEND` and the parent thread checks `pr->ex` only if the return value is `SUSPEND`. At this point, if the corresponding join frame is not allocated yet, the system allocates it and stores the exception in `pr->ex` into it. After this, all threads (including the parent thread) that synchronize at this join point should be stopped as soon as possible. Then the parent thread throws a `stopped` exception immediately (the way how the remaining threads which run on other processors (if any) are aborted is explained in the next section). In the case of slow version code, a thread itself stores an exception into its join frame in `join_to` code.

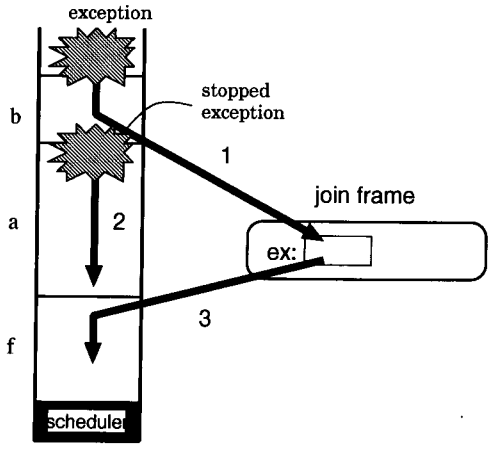
Using the above methods, a join frame may be allocated only for storing an exception.

```
f() {                    a() {
  join {                   par b();
    par a();             }
    par c();
  }
}
```

(a) OPA program



(b) Cactus stack



(c) C stack of the implementation

Figure 7.2: An exception across multiple threads.

```
1   /* create a new thread */
2   pr->ltq_tail = ltq_ptr+1;
3   callee_fr = f__a(pr);
4   if(callee_fr) {
5     if(pr->ex) {
6       (*ltq_ptr)->jf->ex = pr->ex;
7       // throw a stopped exception
8     }
9     f_frame *fr = MAKE_CONT(pr, join_to); callee_fr->caller_fr = fr;
10    if(pr->thief_req) {
11      f_frame *fr = MAKE_CONT(pr, c__a);
12      (*ltq_ptr)->jf = *(pr->js_top); (*ltq_ptr)->jw = SPLIT_JW(pr, pr->jw);
13      (*ltq_ptr)->cont = fr;
14      // save continuation
15      pr->callee_fr = fr; return SUSPEND;
16    }
17  }
18  pr->ltq_tail = ltq_ptr;
```

Figure 7.3: Compiled C code for a fork with support for exception handling.

That is, in this case, there is no thread that has been moved to another processor or has been suspended among the threads sharing the same join target. Here, we examine whether it is possible to delay the allocation of this join frame further. Instead, we store the exception into a fixed place `pr->ex_bak`. The exception is re-thrown after the synchronization that is, in most cases, performed without a join frame (precisely, a join frame may be allocated later because the parent thread that throws a `stopped` exception may be suspended in a certain `finally` block). The problem is that we must always check whether an exception exists not only in join frames (if any) but also in `pr->ex_bak` at exiting every join block. Checking `pr->ex_bak` incurs additional overhead on every join block performed during a program execution. Because we can assume that the number of exceptions thrown during a program execution is much smaller than the number of join blocks performed, it is more efficient to allocate a join frame at the time an exception is actually thrown.

7.2.2 Throwing a `stopped` Exception

When an exception reaches a join block, all threads that synchronize at the join block are stopped as soon as possible. The important point is that they must throw a `stopped` exception on their own so that they can execute `finally` blocks before their termination. If all of these threads are on the C stack where the exception is thrown, only the parent thread have to throw a `stopped` exception as explained previously. By contrast, if some of them have left from the C stack, cooperation with the other processors is necessary.

A running thread periodically checks its join frame's `ex` field by polling. However, checking the immediate join frame that corresponds to the inner most join block is not sufficient for finding an exception. For example, while executing a parallel program pictured as a cactus stack in Figure 7.4, thread 6 (and also thread 8–12) cannot find an exception that has been thrown in thread 7 and stored in join frame C by checking only join frame D (or E, F). As a consequence, each thread must periodically check all join frames that can be found by following the `parent_jf` field iteratively from the immediate join frame (we call this operation an *abort check*).

A drawback of the above way is that, if the nesting level of join blocks is rather deep,

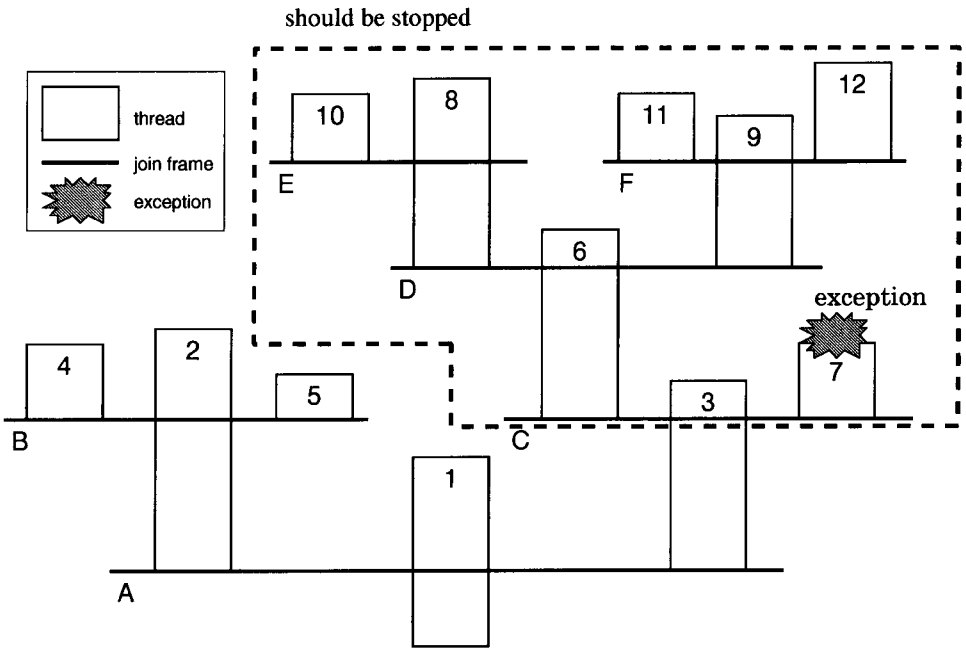


Figure 7.4: Exception stored in a deep-level join frame.

the cost of a single abort check is relatively high. Particularly, the threads irrelevant to the exception (that is, the ones that do not have to be stopped) always check join frames up to the root of the cactus stack. This means that the irrelevant threads suffer from greater overhead. Another drawback is that the overhead is proportional to the frequency of polling, even if the number of the exceptions that are thrown during program execution is very small.

To improve this method, firstly, we prepare a new field `quit` in each join frame, which is used as a flag and is initially set off. The `quit` that is set on means that the whole join block should be stopped. Using this, we can reduce the overhead of a single abort check. In Figure 7.4, when thread 9 finds an exception in join frame C by an abort check, it also sets the flags of join frame D and F on. (For simplicity, the `quit` in join frame C is set on when an exception is stored in it.) After that, thread 11 can notice that it must be stopped by checking only the `quit` flag in join frame F.

In a similar way, we prepare another field, namely `checked`, in each join frame to indicate that the last abort check confirmed that there was no exception below it (and, of course, no exception is set into any join frame below it after the last check).

More precisely, this `checked` field is a *time-stamp*. In this method, time is managed as a global (static) counter of type `int`. Everytime a new exception is stored in a certain join frame, this counter is incremented and the exception is identified (time-stamped) by the value of the counter. By comparing the time-stamp of a new exception and that of join frame(s), each thread can decide wheter it must do abort check now or not.

Next, we show that, instead of polling periodically, the system has only to do abort checks at specific situations. Each thread is usually executed in the state of "no need to do abort check". The precise definition of the state is:

> The last abort check of the thread confirmed that it did not have to stop and, after that, no exception is set into any join frame.

A thread does abort check when one of the following events that make the above condition unsatisfied occurs:

- A new exception is stored in some join frame while the thread is running.

- The thread is resumed. While it was suspended, a new exception may be stored in some join frame.

On the other hand, when a new thread is created as a fast version function call, there is (almost) no need to do abort check. We examine it divided into three cases:

- When the child thread starts its execution, it is in the state of "no need to do abort check." This is because the child thread and the parent thread share the same join target and the parent is in the state of "no need to do abort check" just before the call.

- When the child thread returns normally, the parent is in the state of "no need to do abort check." This is because the child thread and the parent thread share the same join target and the child is in the state of "no need to do abort check" just before the return.

- When the child thread is stopped, the parent must also throw a `stopped` exception because they share the same join target. If the `stopped` exception that the child thread threw for its own use reaches the check code for the `par` call (Figure 7.3, lines 4–17), the parent throws an `stopped` exception by the code of Figure 7.3 (line 7) automatically, i.e., without explicit abort check. Only in the case that the `stopped` exception of the child thread does not reach the parent, the parent must do abort check just after the return. Note that this abort check is done only if the return value is `SUSPEND` because only when the child thread is suspended after it throws an `stopped` exception, the `stopped` is removed from `pr->ex`. So, it does not incur more overhead in the case of normal return.

A new exception is broadcast to all processors if the exception is stored into some join frame. So, it may be broadcast even though no thread that synchronizes at the join point moved to other processors. It may be broadcast even though all threads that moved to other processors have already synchronized at the join point. To avoid these situations, it is broadcast only if the weight value of the join frame where a thread stores an exception is *not* equal to the weight of the weighted reference that the thread has.

Precisely, the entity that is broadcast to all processors is not a (reference to) exception object itself but its time-stamp. Each thread keeps the time-stamp when it did the last abort check in its thread object. By comparing these two time-stamps, a thread can distinguish a new exception from the ones for which it has already done abort check, so it can avoid unnecessary checks.

Chapter 8

Related Work

In this chapter, we discuss some of the previous work related to the theme of this thesis.

8.1 Language Design

8.1.1 Java

Parallel processing in Java [10] is described using the `Thread` class. A new thread is created by creating a new `Thread` object. The `join` operation is provided as a `join` method on the `Thread` object. As was described in Section 2.1.2, such explicit `join` operation makes the programming complicated. Exception handling in Java is designed to handle an exception within the current thread in which the exception is thrown and not to propagate outside the thread.

The design of OPA is intended to keep possible compatibility with Java; however, the synchronization and the exception handling are extended using syntactic constructs.

8.1.2 ABCL/1

Exception handling in a concurrent object-oriented language ABCL/1 [16] is described as methods for exception messages. In ABCL/1, every concurrent object has a thread of control and it can send a message to a concurrent object to invoke a method (script) on the target object concurrently. When an exception occurs during the execution of a

method of an object which receives a message M, an exception message is generated and sent to the sender or the reply destination of M.

Since ABCL/1 is based on concurrent objects, the exception handling is also dealt with by specifying the behaviors of objects to send/receive an exception message. On the other hand, in OPA, exception handlers can be specified independently of objects; furthermore, related threads can be stopped automatically.

8.1.3 KL1 and Shoen

For a concurrent logic language KL1, "shoen" [3] is used to manage a group of goals. A group consists of all goals which are derived from a single initial goal. Shoens can be nested. When a goal raise an exception, the exception is handled by the shoen. Each shoen has a report stream and a control stream for the communication and it can propagate an exception to the outside of the shoen.

The approaches to the exception handing in KL1 and in OPA are similar. However, a shoen in KL1 is a process with its own I/O and it deals with the internal exception. On the other hand, in OPA, the exception handler deals with the exception for a task possibly with parallel execution.

8.1.4 Qlisp

The earlier Qlisp [8] is intended to describe a variety of parallel processing easily and safely with a small set of language constructs. The approach of the earlier Qlisp is very similar to our scheme; in particular, the design for stopping the related threads was described in the report. [8] In the later Qlisp, [9] lots of constructs are added; in particular, the `qwait` construct serves as the `join` construct in OPA.

8.1.5 Approaches Based on First Class Continuations

In some sequential languages, the rest of computation (continuation) can be reified as a first class continuation. The first class continuations are useful to describe non local exit, exception handling and coroutines.

In parallel languages, without first class continuations, coroutines can be realized by simply using multiple threads of control. Non local exit and exception handling can also be realized by using the `catch-throw` constructs of this thesis. Thus, we think the necessity of first class continuations is small in parallel languages. We think, however, the notion of continuation is important to describe the semantics of parallel languages.

The study by Katz and Weise [17] and the study by Hieb and Dybvig [13, 14] are Scheme-based studies on first class continuations for parallel processing. The study by Katz and Weise proposes a scheme to navigate parallel execution and to obtain the same result in parallel execution with `future` [11] and first class continuations as in sequential execution removing `futures`. The study by Hieb and Dybvig [13, 14] proposes constructs to extract (i.e. capture and remove) a part (subtree) of cactus stack (as in Fig. 3.6) and reify it as a first class datum. The reified subtree can be called at any point. However, their construct does not support `finally` clauses.

8.2 Implementation

There are many multithreaded languages or multithreading frameworks that realize low cost thread creation and/or synchronization with automatic load balancing. We classify these languages/frameworks roughly into two categories. One class is for those that support only restricted parallelism. The other class is for those that supports arbitrary parallelism.

8.2.1 Restricted Parallelism

WorkCrews[29] is a model for controlling fork-join parallelism in an efficient and portable manner. When it creates a new thread, it creates stealable entity, (i.e., task) and continues the parent's execution. If the task is not yet stolen when the parent reaches its join point, the parent calls it sequentially. If the task is stolen, the parent thread blocks while waiting for the stolen task's join. Note that, in this model, once a parent thread calls a child thread sequentially, it is impossible to switch context to the parent thread even if, for example, the child thread blocks. So, this model can only be applicable to well-structured fork-join parallelism.

Lazy RPC[5] is an implementation scheme of parallel function call in a C compatible manner. It uses a similar technique as WorkCrews, and so the same restriction on parallel call.

FJTask[19, 18] is a Java's library for fine-grained fork-join multithreading and its technique is similar to Lazy RPC.

Since WorkCrews, Lazy RPC and FJTask employ restricted (well-structured) fork-join parallelism, each task can be started with the stack which is already used by some other tasks; thus the cost of task creation is comparable to that of object allocation.

Cilk[23, 7] is an extended C language with fork-join constructs and, like OPA, its implementation is a compiler from Cilk to C (and runtime). Its implementation technique is also based on LTC-like work steal. However, in several points, it differs from OPA. First, a join construct is lexically-scoped, and does not support other types of synchronization. These simplify the management of child threads. Second, its base language is C, so it does not provide exception handling. Third, it does not have a `synchronized` construct, that is, there is no need to manage thread identity. Fourth, for work steal, Cilk saves a parent thread's continuation in a heap-allocated frame at every thread creation. Indolent Closure Creation[22] is a variant of Cilk implementation and it employs a polling method similar to OPA for LTC. A different point from OPA is that a victim reconstructs the whole stack from all the tasks except the stolen one before continuing its execution.

As compared with Cilk's lexically-scoped join-destination, we think that OPA's dynamically-scoped one is more applicable. This is because a thread's join-destination can be determined as dynamically as a function's caller (return-destination) is determined, and as additionally as an exception handler (throw-destination) is determined. Furthermore, in Cilk, functions that fork some other threads should be called concurrently to avoid poor load balancing, since the sequential call prevents the caller from proceeding without the completion of all threads forked in the callee.

The lexically-scoped join-destination makes the Cilk implementation simple and efficient with the programming restriction. By contrast, OPA realizes a dynamically-scoped join-destination in an efficient manner using laziness, so it does not impose any restriction on programmers. In addition, it is possible to write "lexically-scoped" style programs

in OPA and the compiler can confirm that these programs conform to lexically-scoped style.

8.2.2 Arbitrary Parallelism

LTC[20] (and message passing LTC[4]) is an efficient implementation technique for Multilisp. Multilisp provides dynamic thread creation and general synchronization through future (and implicit touch) constructs, but, it does not have fork-join constructs and so programmers may need some skill to write correct programs. Stack manipulation for work steal is implemented in assembly level, then limited portability. Employing a polling method for LTC is originally proposed in message passing LTC.

StackThreads/MP[24] is also a stack-based approach for multithreading. It enables one processor to use other processor's stack-allocated frames for work steal. It enables general synchronization without heap-allocated frames. To realize this, it only works on shared-memory multiprocessors, and is implemented by assembly level manipulation of stack/frame pointers.

As compared with these languages, OPA has benefits of both categories: simple fork-join constructs, high portability, and general synchronization. In addition, it supports other advanced features like exception handling, `synchronized` method, and so on.

Chapter 9

Performance Evaluation

9.1 Implementation of Multithread

In this section, we evaluate the performance of our OPA implementation compared with Cilk 5.3.2[23], which is known as a good implementation of fine-grained multithreaded languages on multiprocessors. The configurations of the shared-memory parallel computer for this measurements are shown in Table 9.1.

We ported 5 Cilk benchmark programs (`fib`, `knapsack`, `cilksort`, `matmul`, `heat`), that come with the Cilk distribution, into OPA. We also use a binary tree search program described in Section 3.6 for the measurements of the overhead of exception handling.

9.1.1 Measurement Results

Table 9.2 are measurement results of five programs on the SMP. Also, Table 9.3 shows the number of counts of thread creation, task creation and steal while executing the

Table 9.1: Computer settings.

Machine	Sun Fire 3800
CPU	Ultra SPARC III 750MHz, 8MB L2 cache
Main Memory	6GB
Num of CPUs	6
Compiler	gcc 3.0.3 (with `-O3 -mcpu=ultrasparc` option)

programs on our OPA implementation. In order to measure these counts, we added the code which increments the corresponding counter for each event into the C programs generated by the OPA compiler. In all programs, since the overhead of the additional code for these counts is kept between 2–5% of the whole execution time, its probe effect is negligible. The number of counts of thread creation during the execution of a program is independent of the number of processors. The number of counts of steal (and task creation) in the **heat** program seems much larger than the other programs. Because its overall computational structure is organized as step-by-step computations in which each step is completed with a global barrier, these counts seem to be proportional to the number of steps (in this measurement, approximately 40).

9.1.2 Comparison with Cilk

Figure 9.1 shows speedup results of OPA and Cilk programs relative to sequential C programs, that are made from Cilk programs by eliminating all the occurrence of three keywords: `cilk`, `spawn` and `sync`.

Both OPA and Cilk systems show almost ideal speedups (5–6 for 6 CPUs). This means both systems can efficiently distribute workloads among processors. However, in all benchmark programs, OPA system achieved better absolute performance than Cilk as shown in Table 9.2. These results mean that our OPA implementation incurs less overhead for thread creation and synchronization than Cilk. (In the **heat** program whose thread granularity is not so fine, the absolute execution time is almost the same in both systems.) In `cilksort` and matmul, The difference between two systems is not so remarkable as the difference in `fib` and **knapsack**. This seems to be because `cilksort` and matmul use arrays. More precisely, In Cilk programs:

```
int A[N];
f(&A[N/2]);
```

we can pass the latter part of an array `A[N]` to `f()` as a pointer argument to the corresponding element of the array. In OPA programs, however, arrays are Java's array objects:

```
int A = new int[N];
```

Table 9.2: Absolute execution time (and relative time to C in parentheses).

(sec)

# of PEs		1	2	3	4	5	6
fib(38)	C	3.36	-	-	-	-	-
	Cilk	12.1 (3.6)	6.06	4.17	3.03	2.44	2.02
	OPA	6.21 (1.82)	3.39	2.26	1.70	1.36	1.13
knapsack	C	5.03	-	-	-	-	-
	Cilk	9.05 (1.80)	4.64	3.17	2.36	1.86	1.55
	OPA	7.42 (1.48)	3.84	2.63	1.79	1.35	1.17
cilksort	C	2.29	-	-	-	-	-
	Cilk	3.42 (1.49)	1.72	1.16	0.89	0.72	0.62
	OPA	2.91 (1.27)	1.42	0.99	0.72	0.61	0.52
matmul	C	5.02	-	-	-	-	-
	Cilk	7.74 (1.54)	3.83	2.62	1.93	1.56	1.35
	OPA	7.12 (1.42)	3.48	2.34	1.77	1.42	1.19
heat	C	6.49	-	-	-	-	-
	Cilk	6.58 (1.01)	3.35	2.32	1.77	1.48	1.27
	OPA	6.63 (1.02)	3.31	2.05	1.61	1.38	1.20

Table 9.3: The number of thread creations, task creation, and steal.

# of PEs		1	2	3	4	5	6
fib(38)	thread				63,245,985		
	task	0	91	160	253	423	548
	steal	0	9	19	37	56	79
knapsack	thread				26,839,428		
	task	0	99	223	294	379	608
	steal	0	14	25	39	46	77
cilksort	thread				7,072,482		
	task	0	104	774	360	1,177	1,434
	steal	0	21	176	84	265	321
matmul	thread				23,967,450		
	task	0	82	964	850	1,872	2,646
	steal	0	25	249	163	334	522
heat	thread				171,990		
	task	0	82	450	730	965	1,252
	steal	0	82	489	835	1,113	1,471

```
f(A, N/2);
```

where we must pass (a reference) to an array object itself and an index indicating the latter part separately. Also, an access to an element of the array in `f()` takes more cost (for index calculation) than Cilk. These overhead that is irrelevant to multithreading seems to hide the advantage of OPA programs.

In Cilk, the relative execution time of `fib` to C is 3.6: that is almost the same as the result of the paper[7], 3.63. In the paper [7], the overhead of heap frame allocation is about 1.0 and that of the THE protocol is 1.3. The THE protocol is a mostly lock-free protocol to resolve the race condition that arises when a thief tries to steal the same frame that its victim is attempting to pop.

In OPA, the relative execution time of `fib` to C is about 1.82, and the breakdown of OPA's serial overhead for `fib` is shown in Figure 9.2. The total execution time is smaller than that of Cilk, primarily because the OPA system lazily performs heap frame allocation and the OPA system employs a polling method to resolve the race condition between a thief and its victim; that is, the OPA system only incurs the overhead of

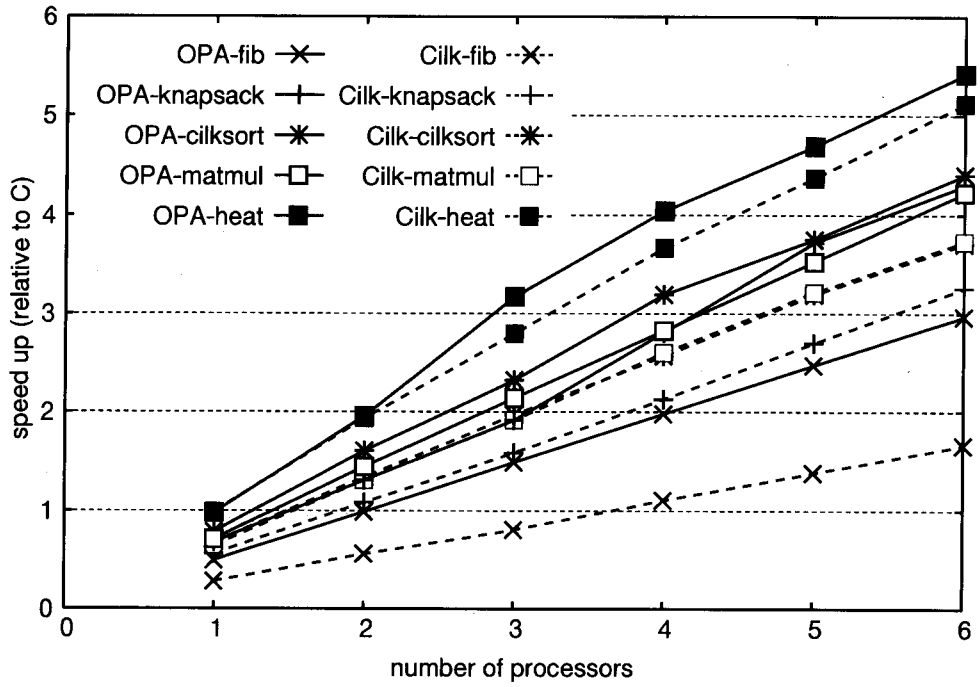


Figure 9.1: Speedup results (relative to sequential C code).

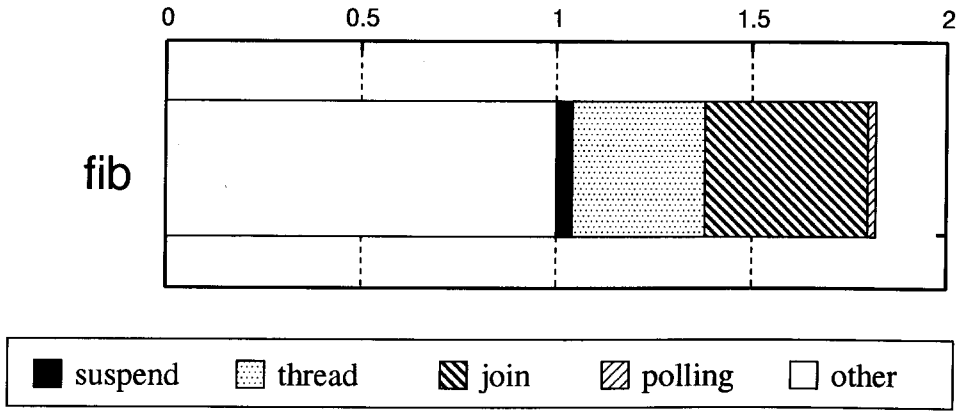


Figure 9.2: Breakdown of overhead for `fib` on a single processor.

Table 9.4: Comparison with the conventional OPA implementation.

(sec)

	A	B	C	D	Cilk
fib(32)	16.3	0.86	0.72	0.37	0.67
fib(38)	292 (estimated)	15.4	12.9	6.8	12.1

suspension check for each method (thread) call, that is about 0.04, and the overhead of polling, that is about 0.02, rather than the overhead of heap frame allocation (stealable continuation creation) plus the overhead of the THE protocol. StackThreads/MP uses the same technique as the OPA system, but it only exhibits almost comparable performance to the Cilk system. This seems to be because the language of StackThreads/MP does not directly permit a thread to return a value or to join to the parent thread; those operations must explicitly be expressed and performed with additional overhead.

In practice, the above evaluation needs to be fixed because of the richer expressiveness of OPA. First, the second recursive call of `fib` can be expressed as a sequential call in OPA, reducing a thread creation cost. Second, supporting advanced features such as thread identification for Java-style locks, dynamically-scoped synchronization, and thread suspension requires additional overhead. More specifically, the overhead of the lazy task queue manipulation (thread) is about 0.34 and that of join stack manipulation and counter check for join synchronization is about 0.42. Even with these additional overhead for the richer expressiveness than Cilk, our implementation of OPA incurs smaller overhead than Cilk by pursuing "laziness".

9.1.3 Comparison with Previous Implementations

Finally, we compared our OPA implementation with laziness to previous OPA implementations using `fib`. To verify the effect of each technique, we prepared four versions of previous implementations:

(a) lazy heap frame allocation for method invocation,

(b) (a) plus lazy task creation (including lazy heap frame allocation for thread creation),

(c) (b) plus lazy creation of thread objects,

(d) (c) plus lazy join frame allocation,

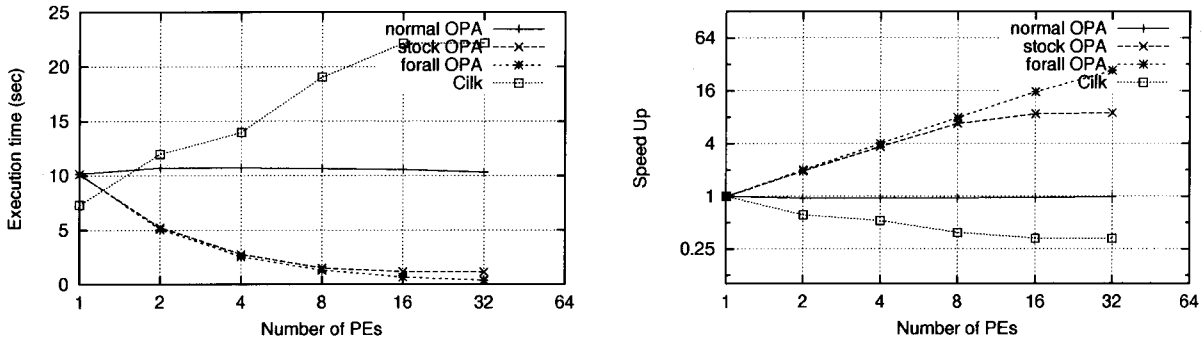
where the later version pursues more laziness than the former ones.

The results are shown in Table 9.4. In implementation (a), since we cannot compute `fib(38)` because of memory deficiency, we also measured `fib(32)`; the value of `fib(38)` in (a) is estimated from the ratios of the other versions and Cilk (they are almost the same, approximately 18). Here, we only give the results on a single processor. Using multiple processors, we can achieve ideal speedups for all versions (and Cilk). First, we can find that the effect of LTC is considerable. This is because the implementation (a) creates a task for each thread creation and because its scheduling policy (that continues the execution of a parent thread at the time of thread creation) does not fit fork-join style parallel processing then causes many context switches. Next, from the results of implementation (b) and (c), we can see that the runtime overhead for supporting advanced features is relatively high. In fact, if manipulations related to these features were not changed to the above ones (Figure 9.2 thread, join), the OPA system could not be a match for the Cilk system.

9.2 `for-par` Loop Execution

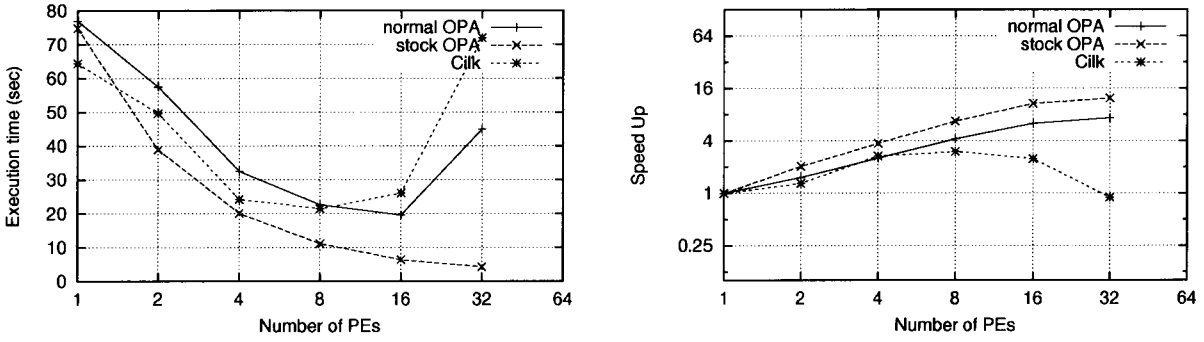
In this section, using two sample programs, **`mandelbrot`** and **`nbody`**, which contain **`for-par`** loop(s), we show that our techniques described in Chapter 6 enable these programs th be executed with good load balancing while normal LTC cannot achieve it. Also, we ported two programs into Cilk in order to compare the performance of the OPA implementation and the Cilk implementation. For this measurements, we used another shared-memory parallel computer: Sun Ultra Enterprise 10000 (Ultra SPARC II 250MHz, 10GB Main Memory, 1MB L2 Cache, 64CPUs).

First, the measurements results of **`mandelbrot`** are shown in Figure 9.3. This program computes the Mandelbrot set within a range of $0 \leq x, y < 1000$. It contains a nested **`for-par`** loop where the inner loop has a single **`par`** call. Using our method, iterations of the loop are divided equally among processors and, unlike simple block distribution, our



(a) Execution Time (sec) (b) Speed up

Figure 9.3: results of `forall`-style `for-par` loop.



(a) Execution Time (sec) (b) Speed up

Figure 9.4: results of non-`forall`-style `for-par` loop.

Table 9.5: Speed up by implementing the abort mechanism.

(msec)

# of PEs	1	2	3	4	5	6
without abort mechanism	0.029	0.223	0.371	0.487	0.597	0.728
with abort mechanism	0.027	0.066	0.093	0.148	0.150	0.158
speedup	1.08	3.35	3.97	3.30	3.97	4.61

method enables to re-divide iterations (repeatedly) so that the whole work of the `for-par` loop, where there is a significant difference in the amount of work among the iterations, can be distributed equally among processors. By contrast, the OPA implementation with normal LTC and the Cilk implementation do not achieve good load balancing since they cannot limit the number of task steals.

Next, the measurements results of **nbody** are shown in Figure 9.4. This program simulates the motion of a number of bodies (for this measurement, $N = 1024$) moving under forces exerted on each by all the others. This program consists of two phases: one is for computing forces acting on each body, and the other is for updating the properties of each body, such as position, velocity, and acceleration. These two phases are both written as a loop which iterates on the set of N bodies. Since the set is represented as a linked list of bodies, there must be an operation to find the next element of the list in each iteration. This means that these loops are non-`forall`-style `for-par` loops. Even in this case, as shown in Figure 9.4, stock mode execution achieves better load balancing results than normal LTC and Cilk.

9.3 Exception Handling

In this section, we evaluate the OPA implementation of exception handling. We use the same computer as in Section 9.1.

We measured two kinds of runtime costs: one is the time spent for aborting threads and the other is the overhead of additional operations by implementing exception handling.

First, we use **nqueens** for the measurement of the time for abortion. It searches

Table 9.6: Execution time to find the first answer.

(sec)

# of PEs	1	2	3	4	5	6
without abort mechanism	6.97	7.00	7.06	7.13	7.20	7.28
with abort mechanism	6.30	6.30	6.36	6.41	6.50	6.56

Table 9.7: Overhead of exception handling.

(sec)

# of PEs		1	2	3	4	5	6
fib(38)	without exception handling	6.81	3.41	2.27	1.70	1.36	1.13
	with exception handling	7.36	3.68	2.45	1.84	1.48	1.24
	overhead rate	1.08	1.08	1.08	1.08	1.09	1.10
14-Queen	without exception handling	3.38	1.72	1.20	0.92	0.76	0.64
	with exception handling	2.71	1.43	0.99	0.77	0.64	0.55
	overhead rate	0.80	0.83	0.82	0.84	0.84	0.86

answers concurrently and, when it finds the first answer, it throws an exception to terminate the whole program. Figure 9.5 shows the **nqueens** method (this method uses bitmaps to represent the configurations of the chess board). We measure the interval between the time when it throws an exception and the time when the exception is caught. The size of the board is 30x30 (`nqueens(30)`). The results are shown in Table 9.5. For comparison, Table 9.6 shows the interval between the time when it starts searching and the time when it finds the first answer.

On a single processor, since no join frame is allocated and no abortion is performed, there is no difference between the two implementations. By contrast, on multiple processors, the time for abortion is reduced obviously. We can also find that, while the time for abortion increases with the number of processors on the implementation without abort mechanism, the time is almost the same among any number of processors on the implementation with abort mechanism. From these results, we guess that more effects can be achieved as we use more processors.

Next, to measure the overhead by implementing exception handling, we use `fib` and the variation of **nqueens** that searches all the answers, i.e., no exception is thrown (the size of the problem is `fib(38)` and `nqueens(14)`, respectively). Since both programs

have no exception handler and throw no exception, we can measure only the overhead by implementing exception handling. The results are shown in Table 9.7.

We can see that the overhead of `fib(38)` is only 8% on average.

Conversely, the implementation of exception handling speeds up the execution of `nqueens(14)`. This is due to optimizations performed by the GCC compiler. In the C code generated by the OPA implementation without exception handling, each method (thread) call is followed by the code for suspension check and the subsequent suspension process. In spite of the low frequency of suspension, the GCC compiler assigns a lot of registers for the suspension process. This prevents the effective assignment of registers for other parts of the code that are executed frequently. By implementing exception handling, an exception check is added at the beginning of each suspension process. This cause the GCC compiler to aware that the frequency of executing the suspension process (following the exception check) is low, then the compiler assigns more registers for other parts than for the suspension process. That is why the implementation with exception handling achieves better performance for `nqueens(14)`.

```
 1 private static void nqueens(int n, int y,
 2 int left, int down, int right) throws Found {
 3   int bitmap, bit;
 4
 5   if(y == n) {
 6     throw new Found(n); // find the answer
 7   } else {
 8     bitmap = MASK & ~(left | down | right);
 9     try {
10       join {
11         while (bitmap != 0) {
12           bit = -bitmap & bitmap;
13           bitmap ^= bit;
14           par nqueens(n, y+1, (left | bit)<<1,
15                       down | bit, (right | bit)>>1);
16         }
17       }
18     } catch(Found e) {
19       e.insert(down); // compose the answer and re-throw
20       throw e;
21     }
22   }
23 }
```

Figure 9.5: nqueens method.

Chapter 10

Conclusion

In this thesis, we proposed efficient and portable implementation techniques for OPA's fork-join constructs. OPA supports several advanced features such as mutual exclusion, Java's `synchronized` method and dynamically-scoped synchronization coupled with exception handling.

Supporting these features has been considered to degrade the effectiveness of existing efficient implementation techniques for fine-grained fork-join multithreaded languages, e.g., lazy task creation. Our implementation techniques pursued "laziness" for several operations such as stealable continuation creation, thread object allocation and join frame allocation.

We compared the OPA implementation with the Cilk implementation. We confirmed that the performance of OPA programs exceeded that of Cilk programs, which indicates the effectiveness of our techniques.

Also, we proposed efficient techniques for good load balancing by which we can divide a loop which contains fork statement(s) in the manner that a victim can remain approximately half of the iterations as its own work. We confirmed that the performance of the OPA system executing such programs scaled up sufficiently with the number of processors.

We also present the efficient and portable implementation techniques of exception handling for the OPA language. By examining the measurement of the time for abortion, we confirmed that parallel search programs can complete their execution faster using our

abort system.

Bibliography

[1] David I. Bevan. Distributed garbage collection using reference counting. In *PARLE: Parallel Architectures and Languages Europe*, number 259 in LNCS, pages 176–187. Springer-Verlag, 1987.

[2] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 6. The MIT Press, 1996.

[3] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, 1988.

[4] Marc Feeley. A message passing implementation of lazy task creation. In *Proceedings of International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, number 748 in LNCS, pages 94–107. Springer-Verlag, 1993.

[5] Marc Feeley. Lazy remote procedure call and its implementation in a parallel variant of C. In *Proceedings of International Workshop on Parallel Symbolic Languages and Systems*, number 1068 in LNCS, pages 3–21. Springer-Verlag, 1995.

[6] Mark Feeley. Polling efficiently on stock hardware. In *Proc. of Conference on Functional Programming Languages and Computer Architecture*, pages 179–190, June 1993.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices (PLDI'98)*, 33(5):212–223, 1998.

[8] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. Technical Report STAN-CS-84-1007, Department of Computer Science, Stanford University, 1984.

[9] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Publishing Company, 1996.

[11] Robert H. Halstead. New ideas in parallel Lisp: Language design, implementation, and programming tools. In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57, Sendai, Japan, June 5–8, 1990. Springer, Berlin.

[12] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[13] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *ACM Conf. on the Principles and Practice of Parallel Programming (PPoPP)*, pages 128–136, March 1990.

[14] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[15] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[16] Yuuji Ichisugi and Akinori Yonezawa. Exception Handling and Real Time Features in an Object-Oriented Concurrent Language. In *Concurrency: Theory, Languages*

and Architecture, volume 491 of *Lecture Notes in Computer Science*, pages 92–109. Springer-Verlag, 1990.

[17] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *ACM Conference on Lisp and Functional Programming*, pages 176–184, June 1990.

[18] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, second edition, 1999.

[19] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM Press, 2000.

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

[21] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of the 1995 conference on Supercomputing (CD-ROM)*, page 41. ACM Press, 1995.

[22] V. Strumpen. Indolent closure creation. Technical Report MIT-LCS-TM-580, MIT, Jun 1998.

[23] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual.* Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA, 2001.

[24] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 60–71, May 1999.

[25] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Extending lazy task creation to support iterative thread creation. In *Proc. of Joint Symposium*

on Parallel Processing 2001(JSPP2001), Kyoto, Japan, pages 157–164, June 2001. (in Japanese).

[26] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Extending lazy task creation for iterative computation. *IPSJ Transactions on Programming*, 43(4):948–957, 2002. (in Japanese).

[27] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Pursuing laziness for efficient implementation of modern multithreaded languages. In *Proc. of 5th International Symposium on High Performance Computing (ISHPC-V)*, pages 174–188, October 2003.

[28] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Lazy normalization techniques for an object-oriented parallel language opa. *IPSJ Transactions on Programming*, 2004. to appear (in Japanese).

[29] Mark T. Vandevoorde. and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[30] Masahiro Yasugi. Hierarchically structured synchronization and exception handling in parallel languages using dynamic scope. In *Proc. of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, July 1999.

[31] Masahiro Yasugi, Shigeyuki Eguchi, and Kazuo Taki. Eliminating bottlenecks on parallel systems using adaptive objects. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques, Paris, France*, pages 80–87, October 1998.

[32] Masahiro Yasugi and Kazuo Taki. OPA: An object-oriented language for parallel processing — its design and implementation —. *IPSJ SIG Notes 96-PRO-8(SWoPP'96)*, 96(82):157–162, August 1996. (in Japanese).

[33] Masahiro Yasugi, Seiji Umatani, Tomio Kamada, Yusuke Tabata, Tomokazu Ito, Tsuneyasu Komiya, and Taiichi Yuasa. Code generation techniques for an object-oriented parallel language opa. *IPSJ Transactions on Programming*, 42(SIG 11 (PRO 12)):1–13, November 2001. (in Japanese).