DATA-DRIVEN DECOMPOSITION OF SEQUENTIAL PROGRAMS FOR DETERMINATE PARALLEL EXECUTION

by

Matthew David Allen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2010

For Heather, my partner in all things.

ACKNOWLEDGMENTS

I am endlessly thankful for my wife Heather and her love and support during graduate school and all my other endeavors. This dissertation is dedicated to her. Heather, I love you. I am also thankful for my wonderful daughters Elizabeth and Chloe, who serve as constant reminders that there is more to life than the ups and downs of graduate school. My parents, David and Terry Allen and Paul and Laurie Wallendal were always there for me with love, support, and advice.

I am deeply grateful to my advisor, Guri Sohi, for guiding me through the research described in this dissertation. Guri constantly encouraged me to tackle challenging, important problems, and to never view a technical problem as insurmountable. His uncanny knack for asking exactly the right questions served to focus my efforts on important aspects of my research that I might have otherwise overlooked.

I have greatly benefitted from interacting with the faculty at UW. My dissertation committee, which included Charles Fischer, Mark Hill, Mikko Lipasti, and David Wood, provided feedback that greatly improved and clarified this dissertation. I am particularly thankful to David Wood for suggesting I investigate a purely software approach to my preliminary research proposal, which led to the work in this dissertation. David was also kind enough to allow me to TA his multi-core programming class, and to use his students as guinea pigs for an early implementation of data-driven decomposition. Ben Liblit gave helpful input before and after my preliminary exam. Susan Horwitz advised me during my first foray into computer science research and publishing, and taught me a great deal about program analysis. I was fortunate to have Jim Smith as my instructor for several architecture courses. I'm not sure I could generate a complete list of all of the faculty that have been generous with their time and advice, but I am grateful to them all.

If I could quantify what I have learned during my time in the department, I'm sure that the lion's share of it could be attributed to my excellent colleagues. Will Benton and Nick Kidd were both my closest friends and most reliable sources of help and advice. Will was extremely generous with his advice on computing, writing, and suggested several improvements to the terminology I used in this dissertation. Nick also served as a fantastic technical sounding board, but I appreciate his sense of humor and endless enthusiasm for every new idea he encountered even more. Jesse Davis has been a great friend who inspired me with his academic excellence and bottomless knowledge of sports trivia. Trey Cain is a another great

friend who could always answer every question about architecture I could pose. Chris Kaiserlian was my friend and partner for many of the courses I took in the early years of graduate school, and taught me a great deal about programming discipline.

My academic siblings were an invaluable resource during my time working for Guri. In the early stages of my research, Adam Butts patiently listened to my naive ideas and helped me to tackle the initial stages of research. My contemporaries, Sai Balakrishnan, Koushik Chakraborty, Jichuan Chang, Allison Holloway, and Philip Wells, were both good friends and stimulating colleagues. I am particularly thankful to Sai and Philip for numerous helpful discussions. My academic little brothers, Srinath Sridharan, Gagan Gupta, and Hongil Yoon have challenged me with great questions, and have provided invaluable feedback on the programming model described in this dissertation.

The students of the Fall 2009 incarnation of David Wood's multi-core programming class also provided great feedback and asked thought-provoking questions about the programming interface for data-driven decomposition. Arkaprava Basu was brave enough to undertake implementing a challenging algorithm as part of his course project, and his experience led to a significant improvement in my runtime system.

Finally, I wish to thank all of the staff of the Computer Sciences Department and the Computer Systems Lab for their support. Angela Thorpe deserves special acknowledgement for helping me navigating the final stages of my dissertation and defense.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

DATA-DRIVEN DECOMPOSITION OF SEQUENTIAL PROGRAMS FOR DETERMINATE PARALLEL EXECUTION

Matthew David Allen

Under the supervision of Professor Gurindar S. Sohi
At the University of Wisconsin-Madison

The transition of the microprocessor industry to multi-core processors poses a significant challenge to the information technology industry. Realizing the full potential of these processors requires decomposition of software into units of work amenable to parallel execution. This places an enormous burden on software developers—conventional approaches to parallel programming, such as multithreading, present a programming model that is radically different from sequential programming, and may result in a wide range of execution behaviors that are neither predictable nor repeatable. These problems threaten to greatly increase the time and cost required to develop software applications.

This dissertation proposes *data-driven decomposition* as a mechanism to overcome many of the difficulties associated with deriving parallel execution of software. Data-driven decomposition dynamically divides a sequential program into units of work according to the data manipulated by its constituent operations. Operations that manipulate disjoint sets of data may execute in parallel, while operations on overlapping data are *serialized*—executed one-at-a-time in program order—to ensure they produce the same result as a sequential execution of the program. Thus data-driven decomposition both preserves both the intuitive sequential programming interface, as well as its predictable, repeatable execution.

This dissertation also describes the design and implementation of PROMETHEUS, a library comprising two components supporting data-driven decomposition of programs written in the C++ language. The first component of PROMETHEUS is a programming interface that allows programmers to express the relationship between computational operations and the data they manipulate via the widespread practices and idioms of object-oriented programming. The second component of PROMETHEUS is runtime support for efficiently orchestrating data-driven decomposition. These mechanisms allow PROMETHEUS programs to realize the benefits of data-driven decomposition, while achieving performance competitive with programs parallelized via conventional, control-driven techniques.

Gurindar S. Sohi

ABSTRACT

The transition of the microprocessor industry to multi-core processors poses a significant challenge to the information technology industry. Realizing the full potential of these processors requires decomposition of software into units of work amenable to parallel execution. This places an enormous burden on software developers—conventional approaches to parallel programming, such as multithreading, present a programming model that is radically different from sequential programming, and may result in a wide range of execution behaviors that are neither predictable nor repeatable. These problems threaten to greatly increase the time and cost required to develop software applications.

This dissertation proposes *data-driven decomposition* as a mechanism to overcome many of the difficulties associated with deriving parallel execution of software. Data-driven decomposition dynamically divides a sequential program into units of work according to the data manipulated by its constituent operations. Operations that manipulate disjoint sets of data may execute in parallel, while operations on overlapping data are *serialized*—executed one-at-a-time in program order—to ensure they produce the same result as a sequential execution of the program. Thus data-driven decomposition both preserves both the intuitive sequential programming interface, as well as its predictable, repeatable execution.

This dissertation also describes the design and implementation of PROMETHEUS, a library comprising two components supporting data-driven decomposition of programs written in the C++ language. The first component of PROMETHEUS is a programming interface that allows programmers to express the relationship between computational operations and the data they manipulate via the widespread practices and idioms of object-oriented programming. The second component of PROMETHEUS is runtime support for efficiently orchestrating data-driven decomposition. These mechanisms allow PROMETHEUS programs to realize the benefits of data-driven decomposition, while achieving performance competitive with programs parallelized via conventional, control-driven techniques.

1 INTRODUCTION

> *I suppose that we are all asking ourselves whether the computer as we now know it is here to stay, or whether there will be radical innovations. In considering this question, it is well to be clear exactly what we have achieved. Acceptance of the idea that a processor does one thing at a time—at any rate as the programmer sees it—made programming conceptually very simple, and paved the way for the layer upon layer of sophistication that we have seen develop. Having watched people try to program early computers in which multiplications and other operations went on in parallel, I believe that the importance of this principle can hardly be exaggerated.*
>
> — Maurice Wilkes (1967)

1.1 MOTIVATION

Nearly five decades ago, Gordon Moore predicted that the number of transistors integrated on a single chip would double every two years (Moore, 1965). Moore's Law has held true to this day, and for most of this duration, computer architects and circuit designers have been able translate this exponential increase in resources directly into improved microprocessor performance. Unfortunately, another of Moore's predictions—that transistor scaling would enable continuously increasing clock speeds for a fixed amount of power per unit area—has not held true. The exponential growth in the number of transistors used to implement modern microprocessors has resulted in a concomitant growth in power consumption and design complexity. Coupled with the diminishing marginal benefits of processor design techniques such as pipelining and superscalar issue, microprocessor designers have been forced to adopt a new approach to harnessing the transistors furnished by Moore's Law (Olukotun and Hammond, 2005). Rather than improving the performance of an individual microprocessor, recent designs integrate multiple processor cores on a single chip.

Multi-core processors provide *parallel execution* as a new avenue for improving software performance. Because software performance may be measured using many different metrics, this improvement may be reflected in a number of ways, including decreased execution time, improved response time, or reduced power consumption. Unfortunately, it is not possible to automatically parallelize most software. As Herb Sutter observes in his famous essay "The Free Lunch is Over"

(Sutter, 2005), to reap the benefits of multi-core processors, software developers must decompose a program into units of computation suitable for parallel execution. Thus multi-core processors have effectively transferred the burden of improving application performance from computer system designers to software developers.

The transition to parallel software development poses an enormous challenge for the information technology industry. Numerous obstacles make parallel programming significantly harder than traditional sequential programming. Namely, software developers must:

1. Identify or develop *parallel algorithms* that decompose the desired computation into a set of potentially independent operations; and

2. Utilize appropriate data structures that promote independent computation.

Surmounting these first two obstacles requires *parallel thinking* (Blelloch, 2009b). Software developers must overcome decades of inertia to refocus their efforts on exposing parallelism in applications. The first component of parallel thinking requires emphasizing parallelism at the algorithmic level. Many widely-used algorithms in current programs are inherently sequential, or are optimized for sequential execution in ways that obscure their computational independence. The process of replacing these algorithms with parallel versions is often difficult, and sometimes impossible. Even if a suitable parallel algorithm already exists, is will usually be significantly more difficult to understand and implement. If such an algorithm does not yet exist, then the programmer must exert time and effort to develop one. Worse yet, some computations are simply not amenable to parallel execution.

The second component of parallel thinking is careful consideration of the data structures used in a program. Because identifying operations on independent data is the key to deriving parallel execution, the organization of data and the selection of container structures has an enormous impact on the amount of parallelism the program exhibits. For example, trees are excellent sources of independence because they recursively partition a data set. By contrast, linked lists often inhibit parallelism, because they encode a total ordering on each list node.

We believe these first two obstacles represent the fundamental challenges to the success of multi-core processors. Establishing the parallel mindset required to overcome these challenges will require a concerted effort in several areas: we must train present and future programmers to understand and foster parallelism; we must develop tools to measure and visualize parallelism; and we must continue

to invent new and improved algorithms that address the growing demand for parallelism.

Conventional parallel programming models further complicate these already formidable challenges. In addition to the first two obstacles, software developers using these models must also:

3. Explicitly identify independence in the static program;
4. Represent this independence using abstractions that radically differ from the abstractions of the sequential programming model, such as the *threads* and *shared memory* of the multi-threaded model, or the *processes* and *messages* of the message-passing model;
5. Identify accesses to *shared data*, and ensure that these accesses are correctly *synchronized* so that concurrent operations do not corrupt this data; and
6. Test and debug the resulting applications, which behave in an unpredictable and unrepeatable fashion due to the arbitrary interleaving of operations from different threads.

The goal of this dissertation is to allow programmers to focus on the former two obstacles by eliminating the latter four. We will argue that the second set of problems occur because conventional parallel execution models employ *control-driven decomposition* that is oblivious to the data manipulated by the operations of the program. The thesis of this dissertation is that a *data-driven decomposition*, which dynamically decomposes a sequential program into series of operations on disjoint sets of data, yields parallel execution that is repeatable, predictable, and provides performance competitive with, or better than, control-driven decomposition.

1.2 OUR CONTRIBUTIONS

The primary contribution of this dissertation is the proposal and development of data-driven decomposition. The key principles of this execution model are: (1) dynamically decomposing a sequential program to identify the data manipulated by each operation, (2) preserving the sequential ordering of operations on a particular data set, and (3) executing operations on disjoint sets of data in parallel. This model has significant implications for parallel execution of software: data-driven decomposition does not admit the notion of shared state, thereby avoiding data races and the need for explicit synchronization. Furthermore, imposing a sequential ordering on all operations manipulating a particular set of

data results in repeatable and predictable program execution. Within the context of data-driven decomposition, we make the following contributions:

Constructs for Data-Driven Decomposition. We propose two constructs for performing data-driven decomposition: *private objects* and *serializers*. A private object is an aggregation of data that is disjoint from other data in the program, which may only be manipulated by the set of operations defined by its specification. Each private object is associated with a serializer, which executes operations manipulating a private object asynchronously, so that program execution may continue before the operations complete, and serializes them so that they execute one-at-a-time, in program order. Private objects provide the disjoint sets of data required for data-driven decomposition, and serializers act in concert to parallelize the execution of operations on different private objects, while each individual serializer maintains the sequential semantics of operations on a particular private object.

Sequential, Object-Oriented Representation. We describe the design and implementation of PROMETHEUS, a C++ library that implements data-driven decomposition in terms of the widespread practice of *object-oriented programming*. Programmers indicate a data-driven decomposition by annotating existing programming constructs such as class specifications and method invocations. By contrast, conventional parallel programming models require programmers to employ new programming constructs with very different semantics from sequential constructs. PROMETHEUS eliminates this obstacle by providing an interface based on sequential programming constructs and dynamically parallelizing program execution in a manner that preserves their semantics.

Serializer Scheduling via Dynamic Task Extension. Task programming systems, sometimes called fork-join frameworks, utilize work-stealing algorithms to perform dynamic scheduling that is provably efficient in terms of both time and space. However, these algorithms assume that tasks are completely independent and can be represented by a static entity in the program text, such as a single function or method. By contrast, a serializer must manage the execution of an arbitrary number of dynamically identified operations that manipulate a particular private object. We describe how to implement serializers in terms of existing task-programming models using *dynamic task extension*. Implementing serializers using tasking primitives allows data-driven decomposition to leverage

the task scheduling support provided by many current libraries and languages, and places serializer scheduling on a firm theoretical foundation.

Library Support for Unrestricted Work-Stealing. We describe the implementation of the PROMETHEUS runtime library, which performs dynamic task scheduling using work-stealing. Because the activations records of fork-join tasks result in a structure that resembles a tree, rather than a stack, previous work-stealing schedulers have required either compiler support for specialized calling conventions, or have restricted the scheduling algorithms to deal with the limitations of linear stacks. PROMETHEUS uses a novel stack allocation and recycling scheme to implement unrestricted work-stealing as a library.

Receiver Disambiguation. Data-driven decomposition requires each potentially parallel operation to identify its *receiver*—the private object it will manipulate—so that it may preserve the sequential ordering of operations on each particular private object. One way to satisfy this ordering requirement is to execute the receiver identification phase of each operation sequentially. However, sequential receiver identification may result in an unacceptable sequential bottleneck for some algorithms. This observation leads us to recognize the *receiver identification problem*, an inherent limitation of any parallel execution model that maintains program ordering of operations on a given data structure. To address this problem, we propose *receiver disambiguation* as a mechanism to allow operations to immediately begin parallel execution, while preserving the sequential order in which they manipulate a particular private object. We describe the design and implementation of the *receiver disambiguation queue*, and apply it to the implementation of divide-and-conquer parallel loops.

This dissertation expands on our previously published work (**?**Allen et al., 2009) in several ways: First, we provide a generalized notion of data-driven decomposition, and a detailed comparison with conventional, control-driven decomposition techniques. Second, we provide a simpler, more general programming model that is more closely tied to object-oriented programming. Third, we describe a more advanced runtime system that supports both nested parallelism, as well as dynamic scheduling. Fourth, our description of the receiver identification problem and receiver disambiguation is entirely new.

1.3 SYNOPSIS

In the remainder of this document, we:

1. Review the abstractions of multithreading and show how it is used to perform a control-driven decomposition, and demonstrate how this results in unpredictable and unrepeatable parallel execution (in Chapter 2);

2. Propose data-driven decomposition as a mechanism to realize repeatable, predictable parallel execution that preserves the intuitive sequential programming model, and describe the application programming interface of PROMETHEUS, a C++ library for performing data-driven decomposition using object-oriented programming (in Chapter 3);

3. Describe an implementation of serializers that enables data-driven decomposition to leverage efficient dynamic scheduling techniques developed for control-driven decomposition (in Chapter 4);

4. Present the design and implementation of the PROMETHEUS runtime library, which performs unrestricted work-stealing to dynamically schedule parallel tasks (in Chapter 5);

5. Explain how the ordering requirements of data-driven decomposition lead to the receiver identification problem and describe the implementation of the receiver disambiguation queue, showing how it can alleviate this problem in the context of parallel loops (in Chapter 6); and

6. Evaluate the performance of data-driven decomposition in comparison with control-driven decomposition of the same applications (in Chapter 7).

Before concluding, we compare data-driven decomposition with other means of realizing parallel execution. Finally, we summarize the insights we have acquired during the development of data-driven decomposition, and discuss their implications for future parallel execution models. We also discuss future research opportunities indicated by the findings of this dissertation.

2 BACKGROUND

> *A folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.*
>
> — Edward Lee (2006)

The sequential execution model is arguably the most successful abstraction in the field of computer science, serving as the basis for most existing software. Three essential features explain this success. The first essential feature is the programming model: sequential programs express computation as a series of operations on a set of variables, each operation completing before the next one begins. This provides an intuitive interface that allows the programmer to reason about each step of the program in isolation, assuming all previous steps have completed, and no subsequent steps have begun.

The second essential feature of the sequential execution model is the *deterministic* manner in which programs execute. The static representation of the program dictates a total ordering on all program operations to which the execution must adhere. Consequently, sequential program execution is *repeatable* because every instance of the program running with a particular input yields the same sequence of operations. Repeatability allows the programmer to easily reproduce errors, and is the basis for most current testing and debugging methodologies. However, repeatability does not fully describe the advantages of encoding a deterministic ordering in a program, because it only captures the behavior of multiple runs of the program. Sequential execution is also *predictable*, because the programmer can reason about what will happen when the program is run with a particular input. The ability to anticipate what a program will do makes it much easier for the programmer to write code that will achieve the desired outcome. Together, the properties of repeatability and predictability provide the programmer with strong guarantees about the runtime behavior of a program.

The third essential feature of the sequential execution model is that, until recently, it has been amenable to high-performance execution that has tracked the exponential transistor scaling of Moore's Law (Moore, 1965). Smith and Sohi (1995) describe the various techniques used to implement the high-frequency superscalar microprocessors that have achieved this performance. However, in recent years, technological forces such as power consumption and design complexity

have conspired to limit further increases in processor performance (Olukotun and Hammond, 2005). Computer architects have therefore changed course, utilizing the exponentially increasing number of transistors at their disposal to integrate multiple processor cores onto a single chip (Sutter, 2005).

Multicore processors have become ubiquitous, offering benefits such as increased performance, improved response time, and decreased power consumption. To leverage these benefits, software must be capable of dividing its constituent computations among the cores of a multicore processor to achieve *parallel execution*. Unfortunately, there is no general way to achieve parallel execution of sequential programs—automatic parallelization has proven largely ineffective, due to the limitations of program analysis, and the fact that sequential programs frequently employ inherently sequential algorithms (Sutter and Larus, 2005). Therefore, to exploit the potential of multicore processors, programmers must intervene and divide up the work of a program in a manner suitable for parallel execution.

Today, the most common technique for realizing parallel execution is *control-driven decomposition*—statically dividing a program into multiple units of work based on independent control flow. The prevalent programming model for this kind of decomposition is *multithreading*, which provides two primary abstractions: multiple *threads of control*, and a global *shared memory*. In contrast with the deterministic ordering of operations in sequential program execution, multithreaded execution is *nondeterministic*. When run, the program imposes no ordering on operations from different threads; instead, the interleaving of these operations is determined dynamically when the program executes. Dynamic ordering makes multithreaded program behavior unpredictable, increasing the difficulty of implementing programs to achieve a desired result.

Control-driven decomposition assigns operations to threads statically, and is usually oblivious to the data these operations will access. Therefore when the program is run, it is often the case that operations assigned to different threads will need to share data. Programs that do not require shared data—so-called *embarrassingly parallel* programs—are trivial to parallelize, but client-side programs rarely exhibit this form of parallelism (Sutter and Larus, 2005). Manipulation of a shared variable by operations in different threads, which are unordered with respect to each other, results in a condition called a *determinacy race* (Netzer and Miller, 1992). Determinacy races cause the values assigned to a shared variable to depend on the dynamic ordering of operations from different threads. Since this ordering may vary on every run of the program, determinacy races compromise the repeatability of the program. This greatly increases the difficulty of detecting and reproducing bugs, consequently increasing the cost of software development

and degrading software reliability.

Multithreading is a natural abstraction of multiprocessor computer systems, but as a programming model, it has grave deficiencies. The programming interface is radically different from that of sequential programs, requiring new constructs such as threads and locks, and the shared memory abstraction changes the semantics of memory operations. There is a broad consensus among computer scientists that multithreaded programming is significantly more difficult than sequential programming (Sutter and Larus, 2005). The difficulty of multithreaded programming is not simply a lack of programmer acumen, but is a fundamental property of the model, demonstrated by the fact many machine-automated static analyses commonly used for sequential programs are provably more complex for multithreaded programs. For example, Ramalingam (2000) shows that context-sensitive static analyses—i.e., analyses that consider only valid inter-procedural paths when analyzing a program—are rendered undecidable if the analysis must also consider the synchronization of a multithreaded program. Lee (2006) makes a convincing argument that the state-space explosion of multithreaded program behavior renders multithreading untenable as a programming model for client-side applications. He predicts that if threads become the de facto standard for programming multicore processors, the resulting proliferation of concurrency bugs will make multicore software extremely unreliable.

Bocchino et al. (2009a) observe that control-driven decomposition is only required for a subset of all parallel programs—*reactive* computations that exhibit inherent concurrency, such as a web server. *Transformative* computations, which manipulate input data to produce a desired output, have no innate concurrency requirement. Most currently-sequential client-side applications fall into this category. While they do not require concurrency, parallel execution of transformative computations allows them to leverage the benefits of multicore processors. The profound difficulty of writing and debugging multithreaded programs motivates the search for new parallel execution models to address this class of computation.

The remainder of this chapter expands on these issues, using as simple bank transaction processing program, presented in Section 2.1, as a running example. Section 2.2 shows how multithreading is used to perform a control-driven decomposition of the example program. Section 2.3 demonstrates how shared data leads to two kinds of *data races*: *atomicity violations*, which may cause operations to compute erroneous results, and *determinacy races*, which cause program execution to be unrepeatable. Section 2.4 considers repeatability, arguing that while determinism is not a realistic goal for parallel execution, a weaker property called *determinacy* provides sufficient repeatability. Section 2.5 explains how de-

```
1  class account_t {
2  private:
3      const unsigned int number;
4      float balance;
5
6  public:
7      account_t (unsigned int number, float balance) :
8          number (number), balance (balance)  {}
9
10     unsigned int get_number () const { return number; }
11
12     float get_balance () const { return balance; }
13
14     void deposit (float amount) {
15         balance += amount;
16     }
17
18     void withdraw (float amount) {
19         balance -= amount;
20     }
21 };
```

Figure 2.1: C++ code for a bank account class

terminacy can be made predictable using the ordering information encoded in a sequential program. Finally, Section 2.6 illustrates the resulting property, which we call *sequential determinacy*.

2.1 RUNNING EXAMPLE: BANK TRANSACTION PROCESSING

This section presents a simple bank transaction processing program to serve as a running example for this dissertation. Rather than representing realistic bank transaction processing, this example is intended to capture many of the problems that arise when parallelizing a program. Our program executes the bank transactions as a transformative computation, whereas a practical system would likely run in real-time and require a reactive, truly concurrent solution. Furthermore, the bank account operations are too fine-grained to be profitably parallelized on current multiprocessors. While unrealistic, we use this example because the familiar bank account idiom allows us to rely on the reader's intuition for what should happen as the program executes.

```
1  // Read bank transactions one at a time,
2  // until there are no more transactions.
3  for (trans_t* trans = get_next_trans (); trans != NULL;
4       trans = get_next_trans ()) {
5      account_t* account = trans->account;
6
7      if (trans->type == DEPOSIT)
8          account->deposit (trans->amount);
9
10     else if (trans->type == WITHDRAW)
11         account->withdraw (trans->amount);
12
13     else if (trans->type == BALANCE)
14         trans->balance = account->get_balance ();
15 }
```

Figure 2.2: C++ bank transaction processing

The code for a bank account class is listed in Figure 2.1. We use the standard C/C++ convention of appending `_t` to differentiate the names of types from the names of variables.[1] Each bank account comprises two fields: a constant integer for the account number (line 3) and a floating-point balance (line 4). The account's interface provides five operations. The constructor (lines 7–8) initializes the account data to the specified values. Two accessor methods return the account number (line 10) and balance (line 12). Two mutator methods allow for deposits (lines 14–16) and withdrawals (lines 18–20) by incrementing and decrementing the balance, respectively.

The bank transaction processing code is given in Figure 2.2. This code reads transactions and processes them in a loop, until there are no further transactions (lines 3–4). First, the loop body reads the account upon which it will operate out of the transaction object (line 5). Second, the loop body determines the type of the transaction and invokes the appropriate method on the account object (lines 7–14).

There are several aspects of the execution of this program that are not known by the programmer at the time the program is written, because they are determined by the input data. The number of transactions in the input determines the number of times the loop will execute. The input also determines which account objects

[1] Java uses a different convention that capitalizes type (class) names to differentiate them from variable names.

Number	Account	Type	Amount ($)
1	1	deposit	40
2	4	withdraw	35
3	1	withdraw	30
4	3	withdraw	10
5	2	deposit	50
6	4	deposit	70
7	1	withdraw	75
8	1	deposit	55
9	4	withdraw	45
10	3	deposit	60
11	4	deposit	25
12	3	withdraw	90

Table 2.1: Example input for bank transaction program

the program will modify, as well as the operation it will perform on the account.

The statically unknown aspects do not pose a problem for the programmer in the sequential execution model. The statically encoded ordering allows the programmer to anticipate what will happen when the program executes, even before it is run. Consider the input shown in Table 2.1: observing that transaction numbers 2, 6, 9, and 11 manipulate account 4, the programmer can predict that the sequence of values of the balance of account 4 will be $\{100, 65, 135, 90, 115\}$. When program execution deviates from what the programmer expects, the repeatability of the sequential ordering makes debugging a straightforward process—errors are trivially reproduced by rerunning the program with the same input. While they may not pose a problem in the context of the sequential execution model, in the next section we will see that the statically unknown aspects have major implications for the multithreaded implementation.

2.2 CONTROL-DRIVEN DECOMPOSITION WITH MULTITHREADING

Multithreading is the predominant programming model for control-driven decomposition. Threads share most resources, including an address space, and maintain only a small amount of truly private state, including a set of registers,

```
1  // Read bank transactions one at a time,
2  // until there are no more transactions.
3  for (trans_t* trans = get_next_trans (); trans != NULL;
4       trans = get_next_trans ()) {
5      transactions.push_back (trans);
6  }
7
8  // Break the transactions into chunks of equal size,
9  // and assign each chunk to a thread.
10 int trans_per_thread = transactions.size () / NUM_THREADS;
11 thread_info_t thread_info[NUM_THREADS];
12 thread_t tid[NUM_THREADS];
13 for (int i = 0; i < NUM_THREADS; ++i) {
14     thread_info[i].begin = trans_per_thread * i;
15     thread_info[i].end = thread_info[i].begin + trans_per_thread;
16     if (thread_info[i].end > transactions.size ())
17         thread_info[i].end = transactions.size ();
18     thread_create (&tid[i], &process_transactions, thread_info[i]);
19 }
20
21 // Wait for threads to complete before exiting
22 for (int i = 0; i < NUM_THREADS; ++i) {
23     thread_join (tid[i]);
24 }
```

Figure 2.3: Bank transaction processing with threads

a stack pointer, and scheduling information. Many multithreading systems provide mechanisms for so-called *thread-local storage* (TLS), but this an addressing convenience—this state is still accessible by other threads.

Figure 2.3 lists a multithreaded version of the bank transaction processing example assuming a simplified interface similar to POSIX threads (IEEE, 2001). We have performed a control-driven decomposition of the program, statically dividing the transaction input into chunks and assigning each chunk of work to a thread. Since the number of input transactions is unknown when this decomposition is performed, the program must read the entire input into an array before it can begin parceling out transactions to threads. Our example program does this by pushing all the transactions into a global shared array `transactions` (lines 3–6). Next, the program divides the input array evenly among the specified number of threads. Note that because the input data is not known when this division of data is

```
1  // Global array to hold input
2  vector <trans_t*> transactions;
3
4  // Structure to package thread arguments
5  struct thread_info_t {
6     int begin;
7     int end;
8  };
9
10 // Each thread executes this function
11 void* process_transactions (void* arg) {
12     thread_info_t* thread_info = (thread_info_t*) arg;
13     int begin_trans = thread_info->begin;
14     int end_trans = thread_info->end;
15
16     for (int trans_num = begin_trans; trans_num < end_trans;
17          ++trans_num) {
18         trans_t* trans = transactions[trans_num];
19         account_t* account = trans->account;
20
21         if (trans->type == DEPOSIT)
22             account->deposit (trans->amount);
23
24         else if (trans->type == WITHDRAW)
25             account->withdraw (trans->amount);
26
27         else if (trans->type == BALANCE)
28             trans->balance = account->get_balance ();
29     }
30
31     return NULL;
32 }
```

Figure 2.4: Code for transaction processing thread

statically coded, the assignment of transactions to threads is completely oblivious to the accounts that will be manipulated by the transactions. The program then creates a thread to handle each chunk of the array (lines 10–19). Finally, the main thread waits for the threads to complete by using the join operation on each of them (lines 22–24).

Figure 2.4 lists the code for the `process_transactions` function executed by each of the threads. A thread first determines the region of the transaction

```
// Thread 1                          1  // Thread 2
int tmp = account->balance;          2
tmp += amount;                       3
                                     4  int tmp = account->balance;
account->balance = tmp;              5
                                     6  tmp += amount;
                                     7  account->balance = tmp;
```

Figure 2.5: Problematic interleaving of two transactions

array for which it is responsible (lines 12–14). It then processes the transactions stored in this portion of the array, in the same order that the original sequential program would (lines 16–29).

If the sets of accounts manipulated by the transactions in each thread are disjoint, then the multithreaded program will produce the correct final state for each bank account. However, if the input to the program contains more than one transaction operating on any given account, it is unlikely that these transactions would all be assigned to the same thread. Since the control-driven decomposition assigns transactions to threads without considering the accounts they will manipulate, there is a good chance that transactions manipulating a particular account will be assigned to more than one thread.

Sharing accounts among threads introduces a significant problem in the program, because the operations performed by these methods are not *atomic*—they do not take effect instantaneously. Instead, they comprise multiple machine instructions that may be interleaved with instructions from other threads. Consider the two interleaved deposit transactions shown in Figure 2.5, which are presented in a pseudo-code that resembles the primitive operations performed by processor instructions. If this interleaving occurs for deposits on two different accounts, i.e., for different values of `account`, then these computations operate on disjoint data and both accounts will have the correct balance at the end of the computation. However, if the deposits are accessing the same account object, i.e., the `account` pointer is the same, then the deposit performed by thread 1 will be lost. This occurs because thread 2 reads the balance of the account (line 4) before it is updated by thread 1 (line 5), and then thread 2 overwrites the value balance written by thread 1 (line 7). This is one of many possible interleavings of these instructions that would result in one of the operations being lost.

This example illustrates the atomicity problem that arises when a control-

driven decomposition assigns operations on the same data to different threads. In the next section, we will examine how the multithreaded programming model addresses this problem, and further examine the consequences of control-driven decomposition.

2.3 SHARED DATA, SYNCHRONIZATION, AND DATA RACES

Bernstein (1966) recognized that when parallel operations manipulate the same data, they may produce a different result than the sequential execution of the same operations. He defined a set of constraints on how operations access data, now called *Bernstein's conditions*, that guarantees the parallel execution of these operations will be equivalent to their sequential execution. Let $\text{READS}(\text{OP}_i)$ be the set of all variables read by operation OP_i, and $\text{WRITES}(\text{OP}_i)$ be the set of all variables written by operation OP_i. Then the sequential execution $\text{OP}_i \,;\, \text{OP}_j$ is equivalent to the parallel execution $\text{OP}_i \parallel \text{OP}_j$ if the following conditions are satisfied:

$$\text{WRITES}(\text{OP}_i) \cap \text{READS}(\text{OP}_j) = \emptyset \tag{2.1a}$$

$$\text{READS}(\text{OP}_i) \cap \text{WRITES}(\text{OP}_j) = \emptyset \tag{2.1b}$$

$$\text{WRITES}(\text{OP}_i) \cap \text{WRITES}(\text{OP}_j) = \emptyset \tag{2.1c}$$

Bernstein's conditions specify that for correct execution, no operation should read or write a variable that may be written by a logically parallel operation. Violation of these conditions is a *data race*. Netzer and Miller (1992) provide a detailed discussion of data races, as well as a taxonomy that divides data races into two categories: *atomicity violations*, and *determinacy races*.[2] We will examine each category of data race in turn.

To prevent operations from interfering with each other when accessing shared data, multithreaded programs employ *critical sections*—blocks of code the programmer intends to be executed atomically. Critical sections use locks or some other form of synchronization to ensure *mutual exclusion*, so that only one thread may execute a critical section protected by particular lock at any given time. When used correctly, critical sections prevent malignant interleavings that cause operations to produce incorrect results.

Figure 2.6 lists an updated account class that employs critical sections using a

[2] Data race terminology is inconsistent. Netzer and Miller refer to atomicity violations as data races, and determinacy races as general races. We find the former set of terms more descriptive.

```
1  class account_t {
2  private:
3      const unsigned int number;
4      float balance;
5      mutex_t mutex;
6
7  public:
8      account_t (unsigned int number, float balance) :
9          number (number), balance (balance)  {}
10
11     unsigned int get_number () const { return number; }
12
13     float get_balance () const {
14         float ret_val = balance;
15         return ret_val;
16     }
17
18     void deposit (float amount) {
19        balance += amount;
20    }
21
22     void withdraw (float amount) {
23        balance -= amount;
24    }
25
26     void lock () { mutex.lock (); }
27
28     void unlock () { mutex.unlock (); }
29 };
```

Figure 2.6: Bank account class with mutex locks

```
1  // Each thread executes this function
2  void* process_transactions (void* arg) {
3      thread_info_t* thread_info = (thread_info_t*) arg;
4      int begin_trans = thread_info->begin;
5      int end_trans = thread_info->end;
6
7      for (int trans_num = begin_trans; trans_num < end_trans;
8            ++trans_num) {
9          trans_t* trans = transactions[trans_num];
10         account_t* account = trans->account;
11         account->lock ();
12
13         if (trans->type == DEPOSIT)
14             account->deposit (trans->amount);
15
16         else if (trans->type == WITHDRAW)
17             account->withdraw (trans->amount);
18
19         else if (trans->type == BALANCE)
20             trans->balance = account->get_balance ();
21
22         account->unlock ();
23     }
24
25     return NULL;
26 }
```

Figure 2.7: Code for transaction processing thread with locks

mutual exclusion construct called a mutex. Using a single mutex lock for operations on every account would be sufficient to ensure correct execution of the program. However, this would severely limit the available parallelism, because only one thread could perform an operation on any account at a given time. Instead, we use a different mutex for each account object (line 5), so that only one thread may operate on a given account, but multiple threads can operate on distinct accounts (McKenney, 1996). We also add methods to lock (line 26) and unlock (line 28) the mutex. Figure 2.7 updates the transaction processing thread function to use lock an account (line 11) before performing a transaction, and unlocking it (line 22) afterward.

```
1  void account_t::transfer (account_t* to_account, float amount)
2  {
3      this->lock ();
4      to_account->lock ();
5
6      this->withdraw (amount);
7      to_account.deposit (amount);
8
9      to_account->unlock ();
10     this->unlock ();
11 }
```

Figure 2.8: Transfer method

Critical sections must be very carefully deployed to protect access to a shared variable that may be concurrently accessed by another thread. Omitting the synchronization for even one such access can allow multiple threads to simultaneously access the shared variable, causing the first category of data race described by Netzer and Miller:

Definition 2.1. *An* atomicity violation *occurs when a shared variable that is accessed inside a critical section is concurrently accessed by another thread, and at least one of the accesses is a write.*

Correctly synchronizing accesses to shared variables is a hard problem. A recent empirical study by Lu et al. (2008) found that two-thirds of randomly sampled concurrency bugs in real-world programs are due to atomicity violations. Vaziri et al. (2006) describe two reasons why atomicity violations are so difficult to avoid: First, synchronization is associated with the code manipulating the data, rather than the data structure itself. Instead of thinking locally about the consistency properties of a data structure, the programmer must reason globally about every possible access to that data structure. Second, placing every access to a shared variable inside a critical section is often not sufficient to ensure correctness. Many programs assume consistency properties involving more than one variable. Atomicity violations involving multiple variables, sometimes called *high-level data races* (Artho et al., 2003) occur when these consistency properties are not maintained by the synchronization of the program. To avoid high-level data races, the programmer must therefore ensure that any invariant involving multiple variables always holds outside of a critical section.

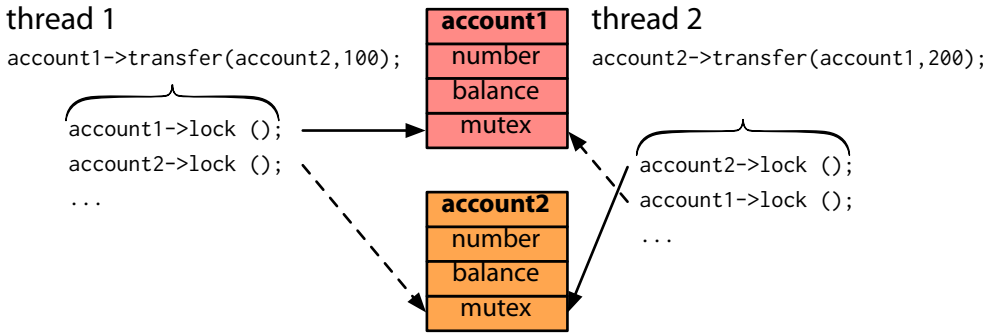


Figure 2.9: Example of deadlock between transfer method invocations

In addition to the difficulty of correctly identifying critical sections, the use of mutual exclusion introduces new types of possible bugs. We present an example of how synchronization can lead to deadlock. Other examples include starvation and priority inversion (Tanenbaum, 2001, chap. 2).

To illustrate a classic example of deadlock, consider the `transfer` method listed in Figure 2.8. This method withdraws an amount from the receiver object and deposits it in the account specified by the first argument. The transfer must maintain a consistent amount of money in the customer's two accounts, so the method locks both accounts before performing both operations (lines 3 and 4). This prevents the transferred amount from seeming to temporarily disappear, which could occur if other threads were allowed to access the accounts involved in the transfer after the withdrawal from the first account (line 6), but before the deposit to the second account (line 7).

Figure 2.9 shows how deadlock can occur if one thread transfers from `account1` to `account2` while a second thread transfers to `account2` to `account1`. Thread 1 successfully locks `account1` and thread 2 successfully locks `account2`. Now neither of the threads can acquire a lock on the second account, since it is held by the other thread, and thus the threads are unable to make further progress. The solution to this problem is to always lock the accounts in a consistent fashion—for example, always locking the account with the lower account number first. In general, deadlock avoidance in multithreaded programs requires strict adherence to complex locking protocols (Coffman et al., 1971).

The synchronization in our multithreaded bank transaction processing ensures that the program computes the correct final balance for each account, but the behavior of the program varies wildly from run to run. Consider running the

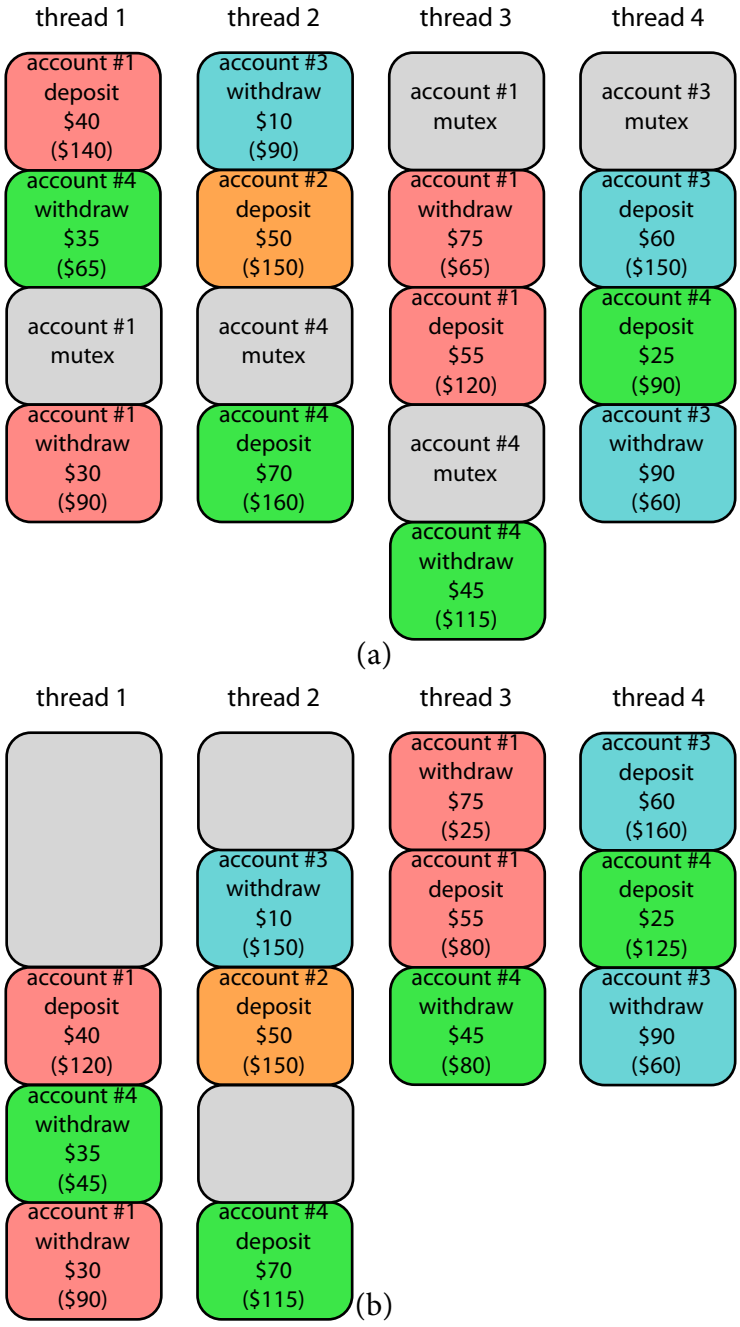


Figure 2.10: Two possible executions of the multithreaded program

program on the input given in Table 2.1 with four threads. Two possible executions are shown in Figure 2.10. (These are not the only possible outcomes of running this program—there are myriad other possibilities.) In both cases, each thread is assigned the same transactions, but when the program is executed they are interleaved in different ways. In the figure, the balance after each transaction is listed in parentheses, and time lost blocking on a mutex is shown as an empty box. We assume the initial balance of each account is $100. During the first execution, shown in Figure 2.10(a), the sequence of balances of account 4 is $\{100, 65, 90, 160, 115\}$. During the second execution, shown in Figure 2.10(b), the sequence of balances of account 4 is $\{100, 125, 80, 45, 115\}$. The final balance is the correct value of $115 in each execution, but the intermediate values are all different. Neither sequence of balances matches the sequential execution, which produces the sequence $\{100, 65, 135, 90, 115\}$ for the balance of account 4.

While the synchronized multithreaded program ensures that each operation leaves the account balance in a consistent state, it does not impose any ordering when operations from different threads access the same account. Thus the values assigned to a particular account balance depend upon how the operations in different threads are dynamically interleaved. This condition is the second category of data race described by Netzer and Miller:

Definition 2.2. *A* determinacy race *occurs when a variable is accessed by multiple threads, at least one of the accesses is a write, and the order of the accesses is not enforced by the program.*

The bank transaction processing example illustrates how static control-driven decomposition leads to execution that is neither predictable nor repeatable. Because the data each operation manipulates is not taken into account during control-driven decomposition, operations that manipulate a particular variable can be assigned to different threads. These operations then result in determinacy races when the program executes, so that the order they update a shared variable depends entirely on the dynamic scheduling decisions of the underlying thread library, operating system, and hardware. The programmer cannot predict this ordering because it is not encoded in the program. And when the program is run multiple times, the ordering will likely change due to varying conditions in the system, so the execution is not repeatable. This makes it extremely difficult to debug multithreaded applications, because so-called "Heisenbugs" (Gray, 1985) manifest in some executions, but not others. Worse yet, latent bugs may be missed entirely during testing, leading to unreliable software that fails in the field.

Repeatability and predictability are both consequences of the ordering properties of a program. Execution is repeatable if the ordering of operations is always the same on different runs with the same input. Execution is predictable if the static program dictates the ordering of program operations. Predictability implies repeatability, because an ordering imposed by the static program will always produce the same execution. The converse is not true—it is possible that running a program with a particular input might always produce the same behavior, but if this order is imposed dynamically, then execution will be repeatable, but not predictable. In the next two sections, we examine how to achieve repeatable and predictable parallel execution.

2.4 REPEATABLE PARALLEL EXECUTION

Researchers often describe repeatability properties of parallel execution using the terms *determinism* and *determinacy*. Unfortunately, previous research provides inconsistent definitions for these terms. And while determinism and determinacy are not synonyms in the fields of mathematics or theoretical computer science, they are often used as such by computer systems researchers. (This confusion possibly results from the fact that determinism and determinacy are near-homophones.) For the purposes of this dissertation, we adopt definitions that illuminate our discussion of repeatable parallel execution, while attempting to maintain consistency with as much existing literature as possible.

We first consider determinism. One popular definition states that a program executes deterministically if it always produces the same output for a given input—see, for example, the definitions given by Bocchino et al. (2009a) and Devietti et al. (2009). This definition is inadequate because it does not consider the intermediate states of the program. As we have shown via the bank transaction processing example, a program may display erratic behavior from run to run, yet always produce the same output for a given input.

In the field of computer science, determinism usually refers to a system that can be modeled as a deterministic finite automaton. To model program execution, the states of a finite automaton would typically be some representation of the program's data, such as the value of all variables, or the contents of processor registers and memory. For such a finite automaton to be deterministic, each state must have a one and only one successor state for each program action. This implies a total ordering on the assignments of values to variables. We prefer the following definition of determinism, which is consistent with automata theory:

Definition 2.3. *A program is* deterministic *if any execution with the same input produces the same totally-ordered sequence of assignments to program variables.*

Due to the cost of interprocessor communication, realizing truly deterministic parallel execution on current multiprocessors is impractical. Ensuring a total ordering of all assignments would require synchronization of all variables, including the private state of each thread. This degree of coordination would cripple the performance of parallel computation. High-performance parallel programs usually only synchronize only accesses to shared state, allowing access to private state to proceed in an uncoordinated, arbitrary order. Therefore any credible model of parallel execution must be nondeterministic to accurately reflect this asynchrony.

Early in the study of parallel computing, researchers recognized need for a weaker repeatability property than determinism. Karp and Miller (1966) give a graph-theoretic model of parallel computation, and in this context define *determinacy* as the property that a variable is assigned the same sequence of values in any execution. Contemporarily, Van Horn (1966) defined the similar notion of *asynchronous reproducibility*. We prefer the former term, which is widely used in academic literature.

Definition 2.4. *A program is* determinate *if each program variable is assigned the same sequence of values in any execution with a particular input.*

We believe that transformative parallel programs should strive to be determinate whenever possible. The fundamental advantage over nondeterminate programs is, by definition, the absence of determinacy races. Because each variable is always assigned the same sequence of values when the program is run with a particular input, determinacy provides the repeatability needed for testing and debugging. In contrast with determinism, determinacy does allow the ordering between assignments to two independent variables to change in different executions. Allowing asynchrony between assignments to different variables relaxes the total ordering of determinism to a partial ordering, facilitating efficient parallel execution.

Determinacy guarantees a reproducible ordering on operations for executions with the same input. However, this property does not specify how this ordering is established. As long as it is repeatable, the ordering may be completely arbitrary. Furthermore, any change in the input to the program admits a completely different ordering of all operations. For example, suppose we modify the input of our bank transaction processing program so that a different set of transactions is performed on account 1, but do not change the transactions on the other accounts. Determi-

nacy allows a different ordering of all operations in the program, even operations on accounts that were not affected by the change in input. So while determinacy provides repeatability, it does not provide *predictability*. We investigate this issue in the next section.

2.5 PREDICTABLE PARALLEL EXECUTION

Program execution is predictable if the programmer has a priori knowledge of its behavior. In contrast with repeatability, which provides guarantees about what happens when a program is run multiple times with the same input, predictability provides guarantees about what how the program will behave before it is executed. Predictability also allows the programmer to reason about the outcomes of running the program with different inputs, which is not addressed by repeatability.

Control-driven decomposition sacrifices predictability because it discards the ordering among operations assigned to different threads. These operations are *dynamically ordered*—they are scheduled according to dynamic decisions made by entities such as the runtime, operating system, and hardware components. By contrast, sequential programs are *statically ordered*—the program text determines the order of all operations, resulting in eminently predictable program execution.

A logical approach to making parallel execution more predictable is to leverage the ordering of a sequential program. We note that while multithreading relaxes the sequential ordering between operations in different threads, it does maintain the sequential ordering for operations in the same thread. Unfortunately this ordering provides little benefit, since it is orthogonal to the ordering of variable assignments needed for repeatability—as we showed in Section 2.3, sharing data among threads introduces determinacy races.

Rather than maintaining the sequential ordering within a unit of independent *control flow* (i.e., a thread), a parallel execution model could maintain the sequential ordering within a unit of independent *data flow*. This yields a stronger version of determinacy where the updates to a variable always occur in the same order as a sequential execution of the program. Rather than requiring a separate sequential version of the program, we define the *sequential elision* as the execution of the program a that elides all parallel constructs. (This notion is equivalent to the *serial elision* described by Frigo et al. (1998).) We then use the sequential elision as the basis for our definition:

Definition 2.5. Sequential determinacy *is the property that in any execution of a program with the same input, a variable is assigned the same sequence of values as*

Number	Account	Type	Amount ($)
1	1	deposit	30
2	2	withdraw	50
3	1	withdraw	70

Table 2.2: Transaction input for determinacy example

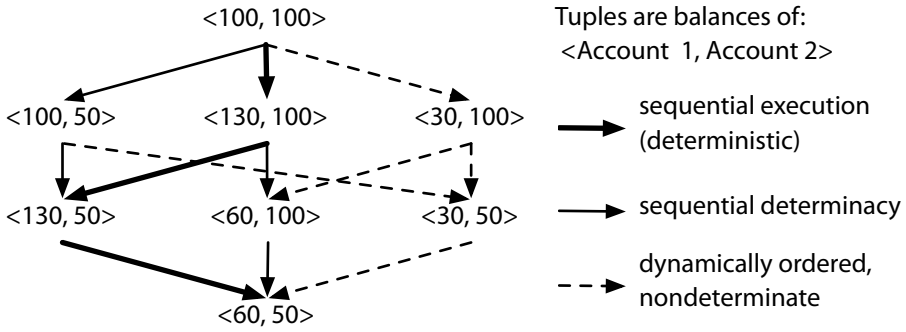


Figure 2.11: Possible executions of the input in Table 2.2

the sequential elision of the program.

Sequential determinacy provides the repeatability of determinacy, because in any execution of the program with the same input, the program will always assign the same sequence of values to a variable. It improves on determinacy by making it predictable, since the ordering is encoded in the program. In the next section, we illustrate sequential determinacy and contrast it with other execution properties.

2.6 SEQUENTIAL DETERMINACY

We illustrate sequential determinacy by revisiting the bank transaction processing example. To restrict the number of possible executions to a manageable size, we use the reduced input shown in Table 2.2. This input includes two transactions on account 1 and one transaction on account 2. Figure 2.11 shows possible program executions under different repeatability and predictability conditions. Each state in the execution of the program is represented as a tuple of the two account balances, which are initially $\langle 100, 100 \rangle$.

The first type of execution depicted in Figure 2.11 is that of a sequential program. Valid sequential transitions are shown by following bold arrows in the figure. Because sequential execution is deterministic, each state has one and only one successor. These state transitions apply the transactions to the accounts in the order given by the input in Table 2.2, yielding a single valid execution: $\langle 100, 100\rangle$, $\langle 130, 100\rangle$, $\langle 130, 50\rangle$, $\langle 60, 50\rangle$.

The second type of execution is that of a sequentially determinate program, indicated in the figure by following any solid arrow in the figure (bold or non-bold). At any given state, any account balance may be updated, but the update must be the next operation on that account in the sequential ordering. Thus from the initial state of $\langle 100, 100\rangle$ there are two valid transitions: applying first transaction to account 1, a deposit of \$30, yielding the $\langle 130, 100\rangle$ state; or the first transaction to account 2, a withdrawal of \$50, yielding the $\langle 100, 50\rangle$ state. However, the withdrawal of \$70 from account 1 is not a valid transition, because it would violate the sequential ordering of transactions on account 1.

The third type of execution is a dynamically-ordered, nondeterminate program, which allows for any ordering of the constituent operations of the program. These executions may follow any arrow (solid or dashed) between states of the program. Note that any execution that applies the second transaction to account 1 before the first falls in this category.

This example illustrates the benefits of sequential determinacy: Programs that satisfy this property rule out any state transitions that produce unrepeatable or unpredictable behavior. Additionally, it allows asynchrony to facilitate parallel execution.

2.7 RELATED WORK

In this section, we discuss alternatives to synchronizing threads with critical sections.

Non-blocking synchronization (Herlihy and Shavit, 2008, chap. 3) employs atomic read-modify-write instructions to implement operations that have a *linearization point*—a single atomic step where the operation takes effect (Herlihy and Wing, 1990). Linearization prevents threads from interfering with each other, and thereby avoids the problems caused when a thread prevents other threads from making progress, such as deadlock and priority inversion. Researchers have used these techniques to develop a number of non-blocking data structures, including queues, lists, hash tables, and trees. Moir and Shavit (2004) provide an excellent survey of the area. Despite these successes, non-blocking synchronization is ex-

tremely difficult to reason about and correctly implement. Universal constructions, such as the one proposed by Herlihy (1991), provide a mechanism to implement any non-blocking algorithm, but they are not practical because they perform poorly, even in comparison with blocking techniques. Therefore non-blocking synchronization will likely remain an important tool for constructing efficient algorithms and data structure, but seems unlikely to by adopted by mainstream programmers.

Transactional memory (TM) is a promising approach for synchronizing multithreaded programs. First proposed by **?** as a programming model for multiprocessors, research on TM has flourished in recent years. **?** provide the authoritative survey of the subject. Transactions comprise a series of operations that appear to take effect instantaneously and indivisibly. They provide three main properties: *failure atomicity*: a transaction either appears completely (i.e., it *commits*), or not at all (i.e., it *aborts*); *consistency*: transactions transition a program from one consistent state to another; and *isolation*: operations in different transactions cannot interfere with each other.

Transactional memory offers programmers several advantages over mutual exclusion techniques. Perhaps the greatest advantage of transactions is that they allow the *composition* of multiple concurrent operations, which is not generally possible using mutual exclusion or non-blocking synchronization. Transactions can also be useful to realize scalable performance without requiring programmers to perform fine-grained locking, which is extremely difficult to do correctly.

While transactional memory addressed many of the problems associated with synchronization, they do not in general address the issue of determinacy races. However, proposals such as TCC (**?**) and IPOT (**?**) introduce the notion of *ordered transactions*, which enforce the sequential ordering of transaction commits. While these models still express parallel programs as a control-driven decomposition, they do ensure sequential determinacy.

2.8 SUMMARY

In this chapter, we attributed the success of the sequential execution model to its intuitive programming model, and two properties of sequential execution: repeatability, which ensures the program behaves the same way on any execution with a particular input; and predictability, which gives the programmer a priori knowledge of a program's behavior. We showed how conventional parallel programs use control-driven decomposition to statically partition a program into units of independent control flow using the multithreaded programming model.

We illustrated how using threads requires constructs not present in the sequential program representation, and how the shared memory abstraction changes the semantics of memory operations. We also showed how control-driven decomposition compromises both repeatability and predictability by assigning operations to different threads that may need to access the same data. The resulting determinacy races produce an explosion of possible program behaviors, making it very difficult for the programmer to reproduce particular behaviors, let alone anticipate what the program will do before it is run.

For the remainder of the chapter, we addressed the question of how parallel program execution could be made repeatable and predictable. We observed that the property of determinacy provides repeatability by ensuring that any variable in the program is assigned the same sequence of values in any execution with the same input. We then showed that determinacy may be made predictable by deriving the ordering of assignments to each variable from a sequential program, yielding a property we call sequential determinacy. In the next chapter, we propose mechanisms to realize parallel execution that satisfies this property.

3 DATA-DRIVEN DECOMPOSITION

> *If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us one without the other?*
>
> — Alan J. Perlis (1982)

In Chapter 2, we argued that many of the difficulties associated with conventional parallel programming are a direct result of control-driven decomposition, which forfeits both the intuitive programming interface and the predictable, repeatable execution of the sequential execution model. We also observed that the abstraction of independent control is unnecessary for transformative computations, which do not require concurrent execution of particular operations. This chapter advocates retaining the the sequential programming interface for transformative computations, and proposes *data-driven decomposition* as a mechanism for deriving repeatable, predictable parallel execution of these programs.

3.1 OVERVIEW OF DATA-DRIVEN DECOMPOSITION

Data-driven decomposition divides up a sequential program for parallel execution based on the data accessed by its constituent operations. Operations that manipulate disjoint sets of data may execute in parallel. Operations that manipulate overlapping data must be *serialized*—executed one at a time, in program order—so that they produce the same effect on the data as would their sequential execution. Serialized operations on a particular set of data may still execute in parallel with operations manipulating different sets of data. Utilizing the knowledge of the data manipulated by each operation allows data-driven decomposition to directly realize Bernstein's conditions when selecting units of work for parallel execution.

Some forms of static parallelization might be viewed as data-driven decomposition. For example, many scientific programs perform computations on large matrices, which can be partitioned so that different regions of the matrix are calculated in parallel. This is possible not because the data is known, but because the regularity of the program's data structures predetermine the data access patterns. By contrast, many client-side applications manipulate pointer-based data structures that are dynamically connected to form irregular graphs. The resulting data access patterns are too unpredictable to be used for static decomposition (Sutter

and Larus, 2005). In general, data-driven decomposition must be performed at run time, when the input to the program is available, and the structure of the data is known. This dissertation proposes *dynamic* data-driven decomposition, although we will usually omit this qualification for the sake of brevity.

We introduce two basic constructs for performing data-driven decomposition: *private objects* and *serializers*. An object comprises a set of data, called *fields*, and defines a set of operations, called *methods*, that may be invoked on the object. A *private object* is an object that is disjoint from other data in the program, and may only be manipulated via its methods.[1] Private objects serve as the disjoint sets of data needed for data-driven decomposition.

Each private object is associated with a *serializer*, which is responsible for the asynchronous execution of methods invoked on the object. A serializer executes these methods one-at-a-time, in the order they were invoked by the program. Serializers thus provide the mechanism for data-driven decomposition to serialize the operations manipulating the data contained in the object.

The programmer facilitates data-driven decomposition in two ways. First, the programmer specifies one or more *classes* of private objects. This specification defines the data stored in objects instantiated from the class, and defines the set of methods that may be performed on these objects. These classes form the basis for partitioning the data of the program into disjoint sets—each class is a specification of a single partition. Objects are instantiated from class specifications at run time to store the data dynamically manipulated by the program. Each new private object provides an additional disjoint set of data that can be used for data-driven decomposition.

Second, the programmer identifies methods that write only data contained in a single private object; and read only the data in that object, or data external to the object that is guaranteed to be constant. We call these *object-pure* methods because their visible side effects are confined to the *receiver* (the object the method manipulates), generalizing the notion of *purity* as the absence of side effects. Previous researchers have defined other relaxed notions of purity (Hogg et al., 1992; Leavens et al., 2006; Sălcianu and Rinard, 2005). Our definition of object-purity is inspired by the classification of Benton and Fischer (2009), which defines an *externally-pure* method as one whose read and write effects are confined to the receiver, and an *externally-read-only* method as one whose write effects are

[1] Here, the term *private* is intended to connote privacy in the sense of object-oriented programming, rather than the multithreaded distinction between shared objects that may be accessed by multiple threads, and private objects that are local to a single thread.

confined to the receiver, but may read external state. Object-pure methods, which may read only constant external data, fall between these categories.

Object-pure methods are *potentially independent* because when two such methods are executed in the program, either they manipulate the same object and are dependent; or they manipulate different objects and are completely independent. By contrast, *object-impure* methods interact with both the receiver object and other state in the program, so they are always dependent. In the bank transaction processing example of Section 2.1, the `deposit` and `withdraw` methods adjust the balance of the account, but do not modify any state external to the object, so they are object-pure methods. The `get_balance` method allows other operations in the program to read the private state of the object, so it is an object-impure method.

Object-pure methods are the candidates for parallel execution in data-driven decomposition. When the running program encounters the invocation of an object-pure method, it *delegates* execution of the method to the serializer associated with the private object. The serializer executes the methods it is delegated serially to honor dependences between method invocations on the same private object. It executes these methods asynchronously with respect to the rest of the program, so method invocations delegated to different serializers present the opportunity for parallel execution.

We illustrate the dynamic actions of data-driven decomposition using the bank transaction processing example from Section 2.1, executing the input given in Table 2.1. We defer detailed discussion of the programming interface until the next section, and assume that the programmer has made the necessary changes to the sequential program to indicate that the `account_t` class is a source of private objects, and that the `deposit` and `withdraw` methods are object-pure. (We will extend the basic model of data-driven decomposition to handle transfers between accounts in Section 3.4.)

Figures 3.1 and 3.2 present several steps in the execution of the example program. In these figures, the values in parentheses indicate the balance of an account after each transaction. Figure 3.1(a) depicts the initial state of program. Four private account objects have been instantiated with a beginning balance of $100, and each account is associated with a serializer.

Figure 3.1(b) shows the actions of the program when it encounters the first invocation of an object-pure method, a deposit of $40 to account 1. Delegation of a method invocation to the appropriate serializer proceeds in three steps, as shown in the figure: ❶ identification of the private object the invocation will manipulate; ❷ identification of the serializer associated with that object; and ❸ communication

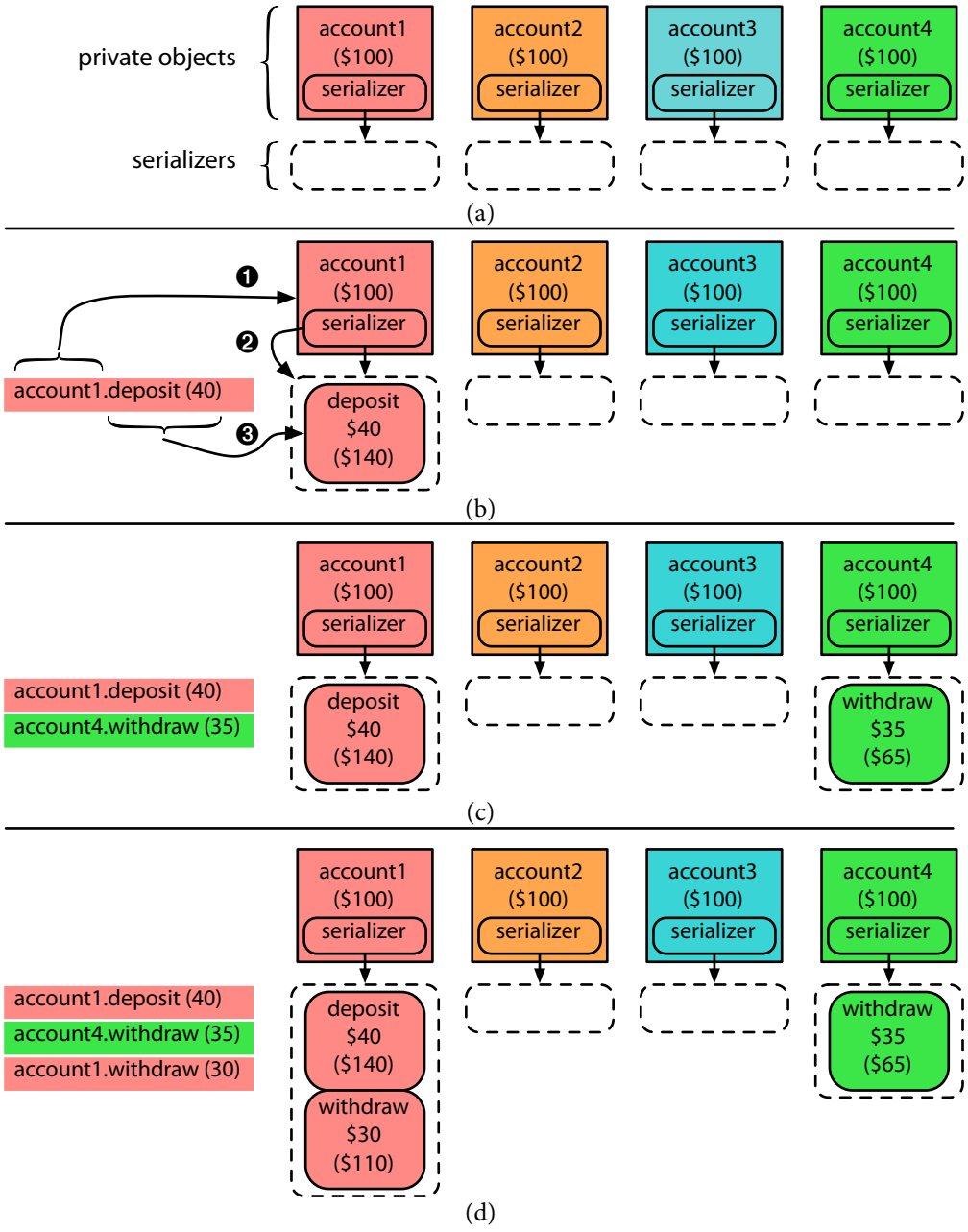


Figure 3.1: Data-driven decomposition with serializers

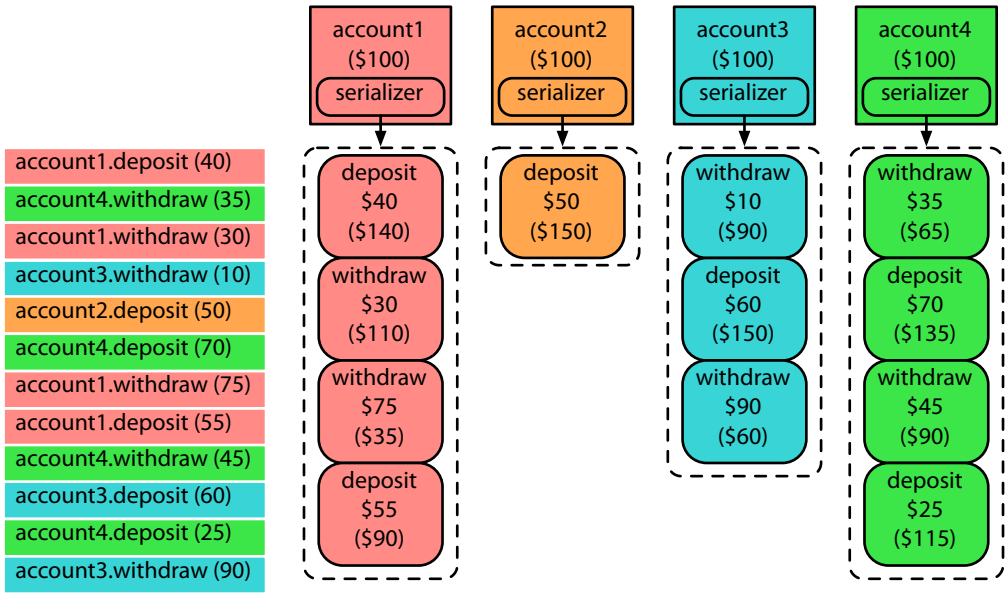


Figure 3.2: Data-driven decomposition with serializers, continued

of the method (`deposit`) and its arguments ($40) to the serializer. The serializer assumes responsibility for executing this method invocation, so the program can immediately proceed to the next operation.

Figure 3.1(c) shows the delegation of the second object-pure method invocation. This transaction performs a withdrawal of $35 from account 4, and is assigned to the serializer for that account. Because this method manipulates a different private account than the previous method, it introduces an additional opportunity for parallel execution.

The third invocation of an object-pure method is a withdrawal of $30 from account 1. As shown in Figure 3.1(d), this invocation is delegated to the serializer of account 1. It is serialized after the first invocation on that account (the deposit of $40), and will not execute until this previous invocation completes and its effects are reflected in the balance. Serialization prevents these two methods from executing concurrently, possibly corrupting the balance of the account, and ensures that the transactions are applied to the account in the order they were encountered in the input to the program.

Figure 3.2 shows the state of the serializers after the program has delegated method invocations for all of the transactions listed in Table 2.1. This example

illustrates two key aspects of data-driven decomposition with serializers. First, serialization ensures that methods are invoked on objects in the same order they would be in the sequential program. Note that as each method invocation is applied to account 4, the sequence of balances is $\{100, 65, 135, 90, 115\}$, exactly matching the sequential execution. Second, the degree of parallelism manifested by the program is determined by the amount of dynamic independence among operations, which is in turn determined by the input to the program. If all the bank transactions were performed on the same account, program execution would be essentially sequential, but if each transaction is performed on a different account, program execution would be highly parallel.

At any given time, a private object may have a number of method invocations pending execution in its serializer, as shown in Figure 3.2. Before invoking a object-impure method, such as `get_balance`, on a private object, the serializer must be *quiesced*. Quiescing the serializer ensures that all asynchronous method invocations delegated to the serializer have completed, so that the state of the object reflects the changes made by these operations. Quiescing the serializer may be performed implicitly any time an object-impure method is invoked, without requiring any extra effort on the part of the programmer.

Implications of Data-Driven Decomposition

Data-driven decomposition is a significant departure from conventional control-driven decomposition. The three distinguishing features of data-driven decomposition are (1) decomposition of a program based on the data accessed by its operations, rather than its control flow, (2) prohibiting, rather than requiring, shared data, and (3) dynamic decomposition instead of static decomposition. We now examine the implications of these differences.

Program Representation. Programming models for control-driven decomposition, such as multithreading, result in code that looks superficially similar to a sequential program. However, these models introduce new constructs like threads and locks that have significantly different behavior than existing sequential constructs. Furthermore, the shared memory abstraction changes the semantics of memory accesses, so that a given piece of sequential code may behave differently in a multithreaded program. By contrast, data-driven decomposition augments the sequential programming model with annotations on existing constructs, including classes and method invocations. Programmers may need to restrict how they use

these constructs to promote parallelism—for example, by limiting the side effects of a method to a single object—but data-driven decomposition does not change their behavior.

Expressing Parallelism. All parallel programming models require the programmer supply a parallel algorithm to realize parallel execution. However, these models differ in how the parallelism of the algorithm is expressed. Multithreading requires programmers to explicitly identify the independent operations in the algorithm and manually assign these operations to threads (or tasks) for execution. Data-driven decomposition takes a different approach, where the programmer expresses dependence in the program via bundling of the individual data elements with the operations that manipulate those elements. Data-driven decomposition then dynamically discovers independence among these operations. While it often behooves the programmer to foster independence with a finer-grained division of data, explicit identification of independence is not a requirement of the model. We believe that dynamic discovery of independence will become increasingly important as multicore processors prompt parallelization of highly irregular applications.

Implicit Synchronization. The shared memory abstraction of control-driven decomposition allows operations in different threads to access the same data. As we saw in Section 2.2, programmers must provide explicit synchronization to prevent operations on shared data from interfering with each other. Data-driven decomposition precludes this problem by forbidding shared data. A private object is the only data that may be manipulated asynchronously, and each private object permits at most one method invocation at any given time. Method invocations may execute in parallel only when they are invoked on different private objects. Therefore each method invocation executes in isolation, and leaves the object in a consistent state. Since parallel method invocations cannot interfere with each other, there are no atomicity violations. Runtime support for data-driven decomposition does rely on synchronization to coordinate parallel execution, but this synchronization is implicit in the program representation.

Repeatable Parallel Execution. Section 2.3 showed that repeatability of a parallel program is compromised when the program exhibits determinacy races, which are unordered accesses to a particular variable in the program. Data-driven decomposition avoids determinacy races by disallowing shared data, and relies on

serializers to enforce the same ordering of method invocations on each private object in any execution with the same input. The only permissible asynchrony results from updates to disjoint sets of data in the program. Data-driven decomposition thus provides determinacy, and the resulting repeatability makes program execution significantly easier to test and debug than a multithreaded program.

Predictable Parallel Execution. In addition to the guarantee that each execution of a program with a particular input always produces the same sequence of method invocations on each private object, serializers also guarantee that this sequence of method invocations will be the same as a sequential execution of the same program. Thus data-driven decomposition ensures the property of sequential determinacy introduced in Section 2.6. For a well-formed data-driven decomposition—i.e., one that enforces the restrictions of private objects and object-pure methods—the equivalence to sequential execution has a very important consequence: all testing and debugging may be performed on a sequential execution, and the parallel execution is guaranteed to produce the same behavior.

Efficient Parallel Execution. Data-driven decomposition dynamically discovers parallelism in a program in the form of method invocations delegated to different serializers. Dynamic runtime support transparently orchestrates parallel execution of the constituent method invocations of each serializer—no additional effort is required from the programmer. This dissertation will show that data-driven decomposition can achieve performance that is comparable to, or better than, control-driven decomposition.

In summary, data-driven decomposition is a means for achieving efficient parallel execution of transformative computations that preserves both the intuitive programming interface and the repeatable, predictable execution of sequential programs. It is not a means for automatic parallelization of sequential programs—programmers must still supply parallel algorithms and appropriate data structures. But unlike control-driven decomposition, data-driven decomposition does not compound these challenges with a radically different programming interface and execution model.

Our prototype implementation of data-driven decomposition is a C++ library named PROMETHEUS. There are two components to this library: the PROMETHEUS template library (PTL) which implements the application programming interface (API); and a runtime system that performs the dynamic actions needed to coordi-

nate program execution. The remainder of this chapter describes the PROMETHEUS API. Chapter 4 describes our implementation of serializers, and Chapter 5 describes the PROMETHEUS runtime system.

3.2 PROMETHEUS: A C++ LIBRARY FOR DATA-DRIVEN DECOMPOSITION

Three primary factors motivated our decision to implement data-driven decomposition as a C++ library. First, a library implementation allows programmers to apply data-driven decomposition in a familiar language, using existing compilers and libraries, and provides a path for parallelizing existing sequential programs. In comparison with a language-based implementation, which would require compiler support, a library implementation also facilitates rapid development and design iteration.

Second, PROMETHEUS closely aligns data-driven decomposition with the *object-oriented programming* (OOP) features of C++. One of the distinguishing features of object-oriented programming is *encapsulation*, which hides the internal implementation of an object and requires manipulation of data in the object to be performed via a set of methods. While OOP is not a prerequisite for data-driven decomposition, encapsulation provides a natural way to derive independent computations on disjoint sets of data.

Third, C++ templates provide a powerful mechanism for *generic programming*—writing classes and functions that are parameterized on types (Stroustrup, 1997, chap. 13). These generic classes and functions can be used to express commonly-used algorithms and data structures in terms of abstract types. When a program uses a template class or function, the compiler *instantiates* the template by replacing the type parameters with the concrete types used in the program, and specializes the resulting code for these types. The compile-time evaluation performed for template instantiation enables *template metaprogramming*—the use of templates as a Turing-complete language for compile-time execution (Veldhuizen, 1995; Czarnecki and Eisenecker, 2000). Appendix A provides an overview the capabilities of C++ templates.

Templates and template metaprogramming provide the necessary tools for implementing language-like features in a C++ library. PROMETHEUS makes extensive use of templates to implement features such as the annotation of private objects and delegation of method invocations. Because template instantiation is performed by the compiler, PROMETHEUS code enjoys the same type checking as ordinary C++ code. PROMETHEUS also employs template metaprogramming to implement compile-time checks for some common programming errors that can

arise during data-driven decomposition.

Data-Driven Decomposition and Object-Oriented Programming

Before describing the PROMETHEUS API, we review the key concepts of OOP as described by Pierce (2002, chap. 18) and Armstrong (2006). Recall that an *object* is a data structure comprising a set of variables called *fields*. Additionally, objects provide an *interface* defining a set of operations called *methods* on the object. Objects thus provide an abstraction of both a set of data, and the operations that modify that data.

A *class* is a blueprint from which objects are instantiated that specifies the number and type of fields in each object, and the implementation of the methods in their interface. Multiple classes may expose the same interface, but provide different implementations of its methods. This enables *polymorphism*—manipulating objects of different types via the same interface. Polymorphic methods are invoked using *dynamic dispatch*, which identifies the correct implementation, or *method body*, corresponding to the method's name and the run-time type of the object.

Polymorphism allows programmers to tailor the behavior of a method based on the type of an object. However, this specialization is often applied to select methods in the interface, with the remaining methods producing the same behavior. Rather than require the programmer to provide redundant definitions of identically behaving methods, OOP provides *inheritance* as a mechanism to incorporate the functionality of an existing class in a new class. When a *subclass* `B` inherits from a *superclass* `A`, `B` includes all of the fields and method bodies of `A`. The subclass may add additional fields and methods to provide additional functionality, and may override the implementation of certain methods in the subclass if a different behavior is desired.

OOP encapsulates data into objects for the purpose of enhancing modularity via *information hiding*—separating the public, abstract interface of an object from its private implementation (Parnas, 1972). Encapsulation alleviates much of the difficulty in developing and maintaining large-scale applications. As long as an object provides a consistent interface, changes to the underlying implementation do not require changes to the software components using that interface. Thus OOP encourages programmers to specify classes so that methods manipulate only the private state owned by an object.

Encapsulation is a means of developing highly modular software, but it also provides the necessary components of data-driven decomposition. Objects inherently provide disjoint sets of data, and methods that operate only on the internal

state of an object are viable candidates for parallel execution any time multiple method invocations operate on different object instances. Implementing data-driven decomposition via OOP further reinforces the practice of encapsulation by providing the dual benefits of increased modularity and increased opportunity for parallel execution.

Besides the logical analogue between data-driven decomposition and encapsulation, there is a practical benefit to closely tying a parallel execution model to OOP. Only an elite subset of current software developers are conversant with conventional parallel programming models. By contrast, object-oriented practices and idioms are already widely used by the software industry. Providing support for data-driven decomposition via OOP should significantly reduce the barriers to parallelizing software and leveraging multicore processors.

The Prometheus API

Having reviewed the basic concepts of object-oriented programming, we now present the PROMETHEUS API, which builds on these concepts to provide support for data-driven decomposition. We begin with the base API, which provides the means to associate C++ objects with serializers and then delegate method invocations to these serializers. Later sections will then expand on this base to describe more advanced capabilities.

Data-driven decomposition requires a mechanism for associating a private object with a serializer. The simplest solution is to incorporate a serializer directly into each private object. This approach would provide a single construct that would be sufficient for many applications of data-driven decomposition.

We anticipate the need for a more flexible mechanism of associating objects with serializers. In particular, the programmer should be able to write code to specify or change the mapping between an object and a serializer. Programmers can then associate multiple objects with the same serializer, effectively treating them as a single private object for the purposes of data-driven decomposition. This is useful when methods operate on a tightly-coupled group of objects, or when a set of objects cannot be broken into disjoint sets of data because they share common data via pointers. Providing the ability to dynamically change the association between objects and serializers allows the programmer the flexibility to express evolving relationships between objects.

To allow dynamic association of private objects with serializers, we propose making the serializer a *first-class object*—an entity that can be assigned to a variable, and passed to and returned from functions. Private objects should then include a

```
1  namespace prometheus {
2      class private_base_t
3      {
4      private:
5          serializer_t* serializer;
6
7      public:
8          // Constructors
9          private_base_t ();
10         private_base_t (serializer_t* serializer);
11
12         // Serializer accessor and mutator methods
13         serializer_t* get_serializer () const;
14         void set_serializer (serializer_t* serializer);
15     };
16 }
```

Figure 3.3: Interface for `private_base_t`

pointer to a serializer, and provide an interface for reading and writing this pointer. The operation responsible for setting the serializer of a private object must quiesce the previous serializer before changing the pointer, to ensure that no more than one serializer is ever executing method invocations on a private object.

PROMETHEUS provides support for associating serializers with C++ objects via the `private_base_t` class, listed in Figure 3.3. This class includes a pointer to a serializer (line 5), a default constructor that automatically creates a new serializer (line 9), and a constructor that allows the programmer to specify a serializer (line 10). The `private_base_t` class also provides a method to get the current serializer (line 13), and a method that sets the serializer to the specified value, after implicitly quiescing the previous serializer (line 14).

Recall the bank transaction processing example from Section 2.1. Figure 3.4 shows how to modify the bank account class of Figure 2.1 so that serializers may be associated with the objects instantiated from this class. This is accomplished by making the bank account class a subclass of `private_base_t` (line 2). (The `public` keyword on line 2 indicates that the public interface of `private_base_t` is visible to users of the `account_t` class.) The constructor for the account (lines 9–11) initializes the account number and balance (line 11) as in the sequential version, and also allocates a new serializer, passing it to the constructor of the `private_base_t` base class (line 10).

```
1  class account_t :
2      public private_base_t
3  {
4  private:
5      unsigned int number;
6      float balance;
7
8  public:
9      account_t (unsigned int number, float balance) :
10         private_base_t (new serializer_t),
11         number (number), balance (balance)  {}
12
13     unsigned int get_number () const { return number; }
14
15     float get_balance () const { return balance; }
16
17     void deposit (float amount) {
18         balance += amount;
19     }
20
21     void withdraw (float amount) {
22         balance -= amount;
23     }
24 };
```

Figure 3.4: Bank account class modified to use serializers

The basic interface of the PROMETHEUS API is given in Figure 3.5. These functions are declared in the `prometheus` namespace to avoid name clashes with identically named functions in user code. The `initialize` (line 3) and `terminate` (line 5) functions create and destroy the PROMETHEUS runtime, respectively. Typically `initialize` will be called once at the beginning of the program and `terminate` will be called once at the end of the program.

PROMETHEUS requires the programmer to indicate regions of code containing delegated method invocations using calls to the `begin_delegation` (line 8) and `end_delegation` (line 10) functions. Calling `begin_delegation` initializes the structures used by the runtime to manage delegation.[2] Calling `end_delegation` effects a *local barrier*, causing program execution to wait until all outstanding

[2] We could avoid the need for `begin_delegation` by having the first delegation perform the initialization, but this would require every delegation to check if initialization is required.

```
1  namespace prometheus {
2      // Initializes the Prometheus runtime
3      void initialize ();
4      // Terminates the Prometheus runtime and cleanup
5      void terminate ();
6
7      // Prepare runtime for delegation
8      void begin_delegation ();
9      // Local barrier: wait for all delegated invocations to complete
10     void end_delegation ();
11
12     // Delegate a method invocation
13     template <typename C, typename B, typename... Args>
14     void delegate (C& obj, void (B::* method) (Args...), Args... args);
15
16     // Wait for delegated method invocations on
17     // obj's serializer to complete
18     template <typename C>
19     quiesce (C& obj);
20 }
```

Figure 3.5: PROMETHEUS API

delegated method invocations have completed.

The calls to `begin_delegation` and `end_delegation` should be lexically scoped so that they clearly delineate a static block of program text. This usage is consistent with the practice of structured programming (Dijkstra, 1972), which makes it easy for the programmer to identify these regions within code. Lexical scoping also ensures that these regions are *properly nested*, so that two different regions are either nested or disjoint, but do not partially overlap. (We will discuss nested delegation in detail in Section 3.3.) With compiler support, it would be straightforward to implement a `delegation{...}` construct to enforce lexical scoping of these regions.

The `delegate` function (line 14) is used to delegate a method invocation to the serializer associated with an object. The arguments to the `delegate` function comprise the object upon which to invoke the method (`obj`), a C++ method pointer (`method`), and the list of arguments (`args`). As described in Section 3.1, delegation identifies the serializer associated with the object, and communicates the object, method, and arguments to the serializer. The serializer executes the method invocations it is delegated one by one, in program order, but asynchronously with

respect to the rest of the program.

The `delegate` function is a template (line 13), parameterized on the type `C` of the object (which must be derived from `private_base_t`), the type `B` of the class in which the method is declared (which must be either `C`, or a superclass of `C`), and a list `Args` of the types of the arguments to the method. This function is an example of a *variadic template*—a template parameterized on a variable number of types (Gregor et al., 2007).[3] Variadic templates allow PROMETHEUS to support delegation of methods with any number of arguments.

When the compiler encounters a call to `delegate` in a program, it uses the template function in the PTL and specializes this generic version with the types used in the program. In turn, this template instantiation uses other template classes and functions to implement the delegation, which are also instantiated at compile time. Because template instantiation is type checked, any errors that would be detected in a standard method invocation—such as invoking an undefined method, or passing an argument of the wrong type—will also be detected for a delegated method invocation.

PROMETHEUS mandates the return value of delegated method invocations be `void`, as shown on line 14. Return values frequently represent a read of object state by the caller (Hogg, 1991), which violates the assumption that the effects of an object-pure method are confined to the receiver object. Return values are also not appropriate for methods executing asynchronously, because they are often used shortly after the method call, inhibiting parallelism between the method's invocation and continuation. To date, we have not encountered a need for return values that outweighs the arguments against them. In practice, it is usually straightforward to refactor a method to store its return value in the object, and provide an accessor method that may be used to retrieve this value later in the program. Should we encounter a compelling application, we anticipate supporting return values by wrapping them in an implicitly synchronized object that causes reads of the return value to wait until the value is produced, similar to the future construct (Baker and Hewitt, 1977; Halstead, 1985).

PROMETHEUS also places restrictions on the types of the parameters of a delegated method. The `delegate` function only accepts methods that take arguments passed by value (i.e., copied), or `const` arguments passed by reference or pointer. In C++, the `const` qualifier on a parameter prevents the method from modifying

[3] Variadic templates are part of a future standard for C++ called C++0x (ISO/IEC, 2010), which as of this writing has not been approved. Many popular C++ compilers, including GCC, have already implemented support for variadic templates.

```
1  begin_delegation ();
2
3  // Read bank transactions one at a time,
4  // until there are no more transactions.
5  for (trans_t* trans = get_next_trans (); trans != NULL;
6        trans = get_next_trans ()) {
7      account_t* account = trans->account;
8
9      if (trans->type == DEPOSIT)
10         delegate (account, &account_t::deposit, trans->amount);
11
12     else if (trans->type == WITHDRAW)
13         delegate (account, &account_t::withdraw, trans->amount);
14
15     else if (trans->type == BALANCE) {
16         quiesce (account);
17         trans->balance = account->get_balance ();
18     }
19 }
20
21 end_delegation ();
```

Figure 3.6: Bank transaction example parallelized with PROMETHEUS

it—any attempt to assign it a value will result in a compilation error. These restrictions on parameter types help ensure that delegated methods meet the requirement that object-pure methods do not write state external to their receiver.

The `quiesce` function (line 19) quiesces the serializer associated with an object. Synchronization ensures that outstanding delegated method invocations have completed, so that an object-impure method may safely execute. This function is parameterized on type `C` of the object (line 18), which must be a subclass of `private_base_t`. The single argument dictates the object to quiesce.

Synchronizing the serializer introduces a subtle issue when a delegated method invocation attempts to quiesce its own serializer. This could potentially cause a method invocation executed by a serializer to wait for that serializer to finish all delegated method invocations, creating a cyclic dependence and resulting in deadlock. This situation is impossible to avoid in some cases, such as when the delegated method invokes another method via a pointer or reference that may refer both to its own receiver as well as other objects. PROMETHEUS automatically detects and elides self-synchronization to prevent this problem.

Figure 3.6 gives the code for a data-driven decomposition of the bank transaction processing program using PROMETHEUS. We do not show the calls to `initialize` and `terminate`, but it should be assumed that they are placed at the beginning and end of the `main` function, respectively. We surround the transaction processing loop with calls to `begin_delegation` (line 1), and `end_delegation` (line 21) to indicate a delegation region.

Recall that because they modify only the balance of the account object on which they are invoked, the `deposit` and `withdraw` methods are object-pure, and thus potentially independent. We delegate these methods as shown on lines 10 and 13. The calls to `delegate` pass the account object (`account`); a C++ method pointer to the desired method (`deposit` or `withdraw`), qualified by the class in which they are defined (`account_t`); and the argument to the method (`trans->amount`).

The `get_balance` method is object-impure, because it allows other operations to read the private state of the account object. Before invoking this method, the serializer associated with the account object must be quiesced, so that all delegated invocations of `deposit` and `withdraw` are reflected in the balance before it is read. Therefore the program invokes `quiesce` on the account object (line 16) before invoking `get_balance` (line 17).

To facilitate data-driven decomposition, the programmer provides a clear delineation between object-pure methods, which are delegated, and impure methods, which are invoked in the standard fashion. Thus there is no reason to require the programmer to explicitly quiesce the serializer of an object. Instead, the serializer can be implicitly quiesced any time an impure method is invoked. PROMETHEUS implements implicit synchronization with the *private wrapper* class.

Implicit Synchronization via the Private Wrapper

The base API of PROMETHEUS shown in Figure 3.5 provides a means for delegating object-pure method calls, and for synchronizing serializers before invoking impure methods. Unfortunately, intermingling these API calls with ordinary method invocations can violate the assumptions of data-driven decomposition in two ways. First, if a programmer mixes standard invocations and delegations of object-pure methods, these method invocations may execute concurrently when they should have been serialized. Second, if the programmer forgets to quiesce the serializer before invoking an object-impure method, that method may execute before outstanding delegated method invocations on the receiver complete. Both of these scenarios introduce the possibility of determinacy races by allowing multiple methods to concurrently manipulate the same object.

```
1  namespace prometheus {
2
3      template <typename C>
4      class private_t <C> : protected C
5      {
6      public:
7          // Constructor
8          template <typename... Args>
9          private_t (Args... args);
10
11         // Delegation of object-pure methods
12         template <typename B, typename... Args>
13         void delegate (void (B::* method) (Args...), Args... args);
14
15         // Quiesced call for object-impure methods
16         template <typename R, typename B, typename... Args>
17         R call ((B::* method) (Args...), Args... args);
18     };
19 }
```

Figure 3.7: Wrapper template interface

PROMETHEUS provides the private wrapper template to forestall these sorts of errors. The private wrapper walls off an object, ensuring its fields are not visible outside of the wrapper (even if public), and mediates all methods invoked on the object to ensure they are either delegated to the object's serializer, or that the serializer is implicitly quiesced before a method is invoked directly. This prevents direct invocation of methods, and eliminates the need for the programmer to explicitly quiesce objects.

Figure 3.7 shows the interface of the private wrapper, defined in the template class `private_t`. This class defines the three permissible operations on a private object. The private wrapper is parameterized on the object's class `C` (line 3), and inherits from `C` using the `protected` keyword (line 4), giving the wrapped object all of the functionality of the underlying object, but hiding its interface from users. This forces the programmer to perform all manipulation of the wrapped object through the wrapper interface.

The first permitted operation is construction (line 9). Wrapped objects may be constructed using the standard constructor interface of the underlying object. Consider an object of the `account_t` class, which could be declared in this fashion:

```
account_t account (account_num, initial_balance);
```

The declaration of a wrapped object takes the same constructor arguments, but uses the type of the wrapped object:

```
private_t<account_t> account (account_num, initial_balance);
```

There is no way to create wrapped objects from unwrapped objects—they can only be constructed inside the boundaries of the private wrapper. This prevents programmers from inadvertently circumventing the `private_t` interface by accessing an object outside of its wrapper.

The second permitted operation is delegation (line 13. The `delegate` method of the private wrapper takes a pointer to a method in the underlying object, followed by a list of arguments, and delegates this method invocation to the object's serializer. Thus:

```
delegate (account, &account_t::deposit, trans->amount);
```

becomes:

```
account.delegate (&account_t::deposit, trans->amount);
```

The third permitted operation is a dependent method call (line 17), which implicitly quiesces the serializer of the underlying object before directly invoking the specified method. The `call` interface of the private wrapper may introduce more synchronization than is strictly necessary, because every dependent call is quiesced, even when no delegation has occurred since the last dependent call. Chapter 4 will show that synchronization is essentially free if a serializer has no outstanding method invocations, so in practice this extra synchronization incurs a negligible performance penalty.

The interface for dependent method calls is similar to delegation, except that it allows return values. As an example, the following dependent method invocation, which must be quiesced using the basic PROMETHEUS API:

```
quiesce (account);
trans->balance = account->get_balance ();
```

requires no explicit synchronization using the private wrapper:

```
trans->balance = account.call (&account_t::get_balance);
```

The constructor, `delegate`, and `call` methods serve as the primary interface to wrapped objects. As a practical matter, the private wrapper also defines the set of operators that C++ allows to be overloaded, and an operator that can be used

```
1  // Declare private wrapper class
2  typedef private_t <account_t> private_account_t;
3
4  begin_delegation ();
5
6  // Read bank transactions one at a time,
7  // until there are no more transactions.
8  for (trans_t* trans = get_next_trans (); trans != NULL;
9        trans = get_next_trans ()) {
10     private_account_t* account = trans->account;
11
12     if (trans->type == DEPOSIT)
13         account->delegate (&account_t::deposit, trans->amount);
14
15     else if (trans->type == WITHDRAW)
16         account->delegate (&account_t::withdraw, trans->amount);
17
18     else if (trans->type == BALANCE)
19         trans->balance = account->call (&account_t::get_balance);
20 }
21
22 end_delegation ();
```

Figure 3.8: Bank transaction code using the private wrapper

to cast the object to wrapped versions of other classes in the object's inheritance hierarchy.

In Figure 3.8, we update the bank transaction processing code to use the private wrapper and its interface. The `account_t` class requires no modification to use the wrapper—we simply define a wrapped type on line 2. We similarly modify the declaration of the account object on line 10 use the wrapped type. We also change lines 13 and 16 to use the `delegate` method to delegate the potentially independent invocations of `deposit` and `withdraw`, and line 19 to use the `call` method for the dependent invocation of `get_balance`.

In summary, the private wrapper protects programmers from errors caused by improper method invocations on private objects by providing an interface that allows only delegations or quiesced calls. The distinction between delegating and calling methods is similar to the distinction that some object-oriented programming languages make between two types of methods: *procedures* that modify an object but do not return a value, and *functions* that return a value but do not

modify the object. Meyer (1988b) referes to this as *command-query separation*. This is yet another example of the synergy between data-driven decomposition and OOP.

3.3 NESTED DELEGATION

Parallel execution models can be divided into two categories: *flat* models and *nested* models (Blelloch, 1996). Flat models allow a single decomposition of a computation to derive parallel operations, but the resulting operations cannot be further decomposed. Nested models allow multiple decompositions of a computation to expose a hierarchy of parallelism, and are generally preferable because many programs exhibit multiple granularities of independence. Blelloch (1996) provides numerous examples of algorithms containing nested parallelism, including Quicksort (Hoare, 1961b), finding primes with the Sieve of Eratosthenes, sparse matrix multiplication, the Quickhull algorithm for computing planar convex hulls (Preparata and Shamos, 1985), and the fast Fourier transform (FFT) (Hillis and Steele, 1986). Before exploring nested data-driven decomposition and its implementation in PROMETHEUS, we review three important arguments for nested parallelism.

The first argument for nested parallelism is that it is the most straightforward way to express recursive computations, such as divide-and-conquer algorithms (Cormen et al., 2009, chap. 27), because it allows the parallelism can be expressed directly in the recursion. Without nested parallelism, the computation must be split into different cases to distinguish the parallel and sequential parts of the recursion.

The second argument for nested parallelism stems from modular programming practices such as OOP, dynamic linking, and dynamic class loading. These techniques facilitate large-scale software development by isolating functionality in modules that interact via abstract interfaces, so that a programmer writing a particular module is often unaware of the implementation details of other modules. Blelloch and Sabot (1990) observe that a particular module may be called by both sequential and parallel code, and similarly the module may call other code that is either sequential or parallel. Ideally, a programmer would express the parallelism in an individual module without needing to consider how parallelism is expressed in other modules, and rely on the parallel execution model to handle any nested parallelism that results from the composition of these modules.

The third argument for nested parallelism is that without it, the programmer must identify the single granularity of parallelism that achieves the best perfor-

mance. This is usually an empirical process, performed either by profiling the application, or applying parallelism at different granularities and measuring performance. While this approach may achieve good performance for certain inputs to the program on a particular hardware configuration, the program may perform poorly under different conditions. By contrast, dynamic task scheduling systems benefit from an abundance of parallelism (Frigo et al., 1998). Using these systems, the programmer can express parallelism wherever it occurs in the program, and the runtime automatically selects the appropriate granularity for parallel execution as the program executes.

Data-driven decomposition exposes parallelism when operations manipulate disjoint sets of data. It exposes nested parallelism when a particular operation on a distinct data set can be further decomposed into operations that manipulate disjoint subsets of that data. Using the constructs proposed in this dissertation, a private object may encapsulate other private objects, each with their own serializer. Method invocations delegated to the serializer of the top-level private object may then perform *nested delegation*—delegating object-pure method invocations to the serializers of the sub-objects. PROMETHEUS supports arbitrarily deep nested delegation.

We demonstrate nested delegation with an implementation of the Quicksort algorithm. The PROMETHEUS code for a data-driven decomposition, given in Figure 3.9, uses the `qsort_block_t` class to sort a region of an array. The program begins with a single block object for the range to be sorted, and recursively subdivides these blocks during the sorting process.

Before the class specification of the `qsort_block_t`, we use a forward declaration the private version, `private_qsort_block_t` (line 2), so we can instantiate private objects of this class inside the class specification. The `qsort_block_t` class inherits from `private_base_t` class to enable association of objects with serializers (line 4). Each object of this class has four fields: the `begin` (line 7) and `end` (line 8) fields, which define the half-open range `[begin,end)`[4] to be sorted, and two pointers to private block objects that may be used to divide this block into a `left` block (line 9) and a `right` block (line 10).

The constructor (lines 13–16) takes two iterators as arguments, which define the range of the block. It creates a new serializer, and associates it with the object by passing it to the constructor of the `private_base_t` class (line 14). The constructor also initializes the `begin` and `end` fields to the values passed to the

[4] A half-open range `[begin,end)` includes the elements `begin`, `begin+1`, ..., `end-1` (but not `end`).

```
1  class qsort_block_t;
2  typedef private_t <qsort_block_t> private_qsort_block_t;
3
4  class qsort_block_t : public private_base_t
5  {
6  private:
7     iterator_t begin;
8     iterator_t end;
9     private_qsort_block_t left;
10    private_qsort_block_t right;
11
12 public:
13   qsort_block_t (iterator_t begin, iterator_t end) :
14         private_base_t (new serializer_t),
15        left (NULL), right (NULL)
16     {}
17
18     void sort () {
19         // find pivot
20         Iter pivot = partition (begin, end);
21
22         begin_delegation ();
23
24         // left side
25         if ((pivot - begin) > 1) {
26             left = new private_qsort_block_t (begin, pivot);
27             left->delegate (&qsort_block_t::sort);
28         }
29
30         // right side
31         if ((end - pivot) > 1) {
32             right = new private_qsort_block_t (pivot, end);
33             right->delegate (&qsort_block_t::sort);
34         }
35
36         end_delegation ();
37     }
38 };
```

Figure 3.9: Recursive data-driven decomposition of Quicksort

constructor (line **??**), and initializes the `left` and `right` block pointers to null (line 15).

The `sort` method (lines 18–37) modifies only data in the range of the array owned by the object, and so it is object-pure. It begins by calling `partition` (line 20), which selects a pivot element; then places all elements less than this element to its left, and all elements greater than or equal to this element to its right; and returns the final location of the pivot element (Hoare, 1961a). Note that the partitioning process requires $\Theta(N)$ comparisons, introducing a significant sequential bottleneck in the algorithm if it is not parallelized. We do not show the implementation of `partition` in this example, but this operation is also amenable to data-driven decomposition using algorithms such as those described by Heidelberger et al. (1990) and Tsigas and Zhang (2003).

The next step of Quicksort splits the array around the pivot element, and then recursively sorts these partitions. Because they are disjoint, we may apply data-driven decomposition to sort the left and right sides in parallel. To enable delegation, this part of the `sort` method is enclosed in calls to `begin_delegation` (line 22) and `end_delegation` (line 36).

The code on lines 25–28 processes the left partition. We first check that the partition contains least two elements (line 25); otherwise, the recursion has completed for this side. If there are enough elements to be sorted, a new `private_qsort_block_t` object is created for the left partition. This new object is encapsulated inside the current object, and represents a subset of its data, i.e. the portion of the array in the range `[begin,pivot)`. This makes it a valid candidate for nested delegation. Construction of the `left` object creates a new serializer, to which we delegate an invocation of `sort` for the left side of the array (line 27).

The code for the right partition is identical to the left partition, except that it sorts the part of the array to the right of the pivot (lines 31–34). When the `sort` method has been delegated for both the left and right partitions, the `end_delegation` method causes execution to wait for these asynchronous methods to complete.

Figure 3.10 depicts the execution of the code from Figure 3.9 sorting the string "PARALLELISM". To perform a sort, we simply declares a block for the range to be sorted, and delegate the `sort` method:

```
private_qsort_block_t block (begin, end);
block.delegate (&qsort_block_t::sort);
```

Initially, the entire array is owned by a single block object associated with serializer_0. For the purposes of this example, we assume that the rightmost ele-

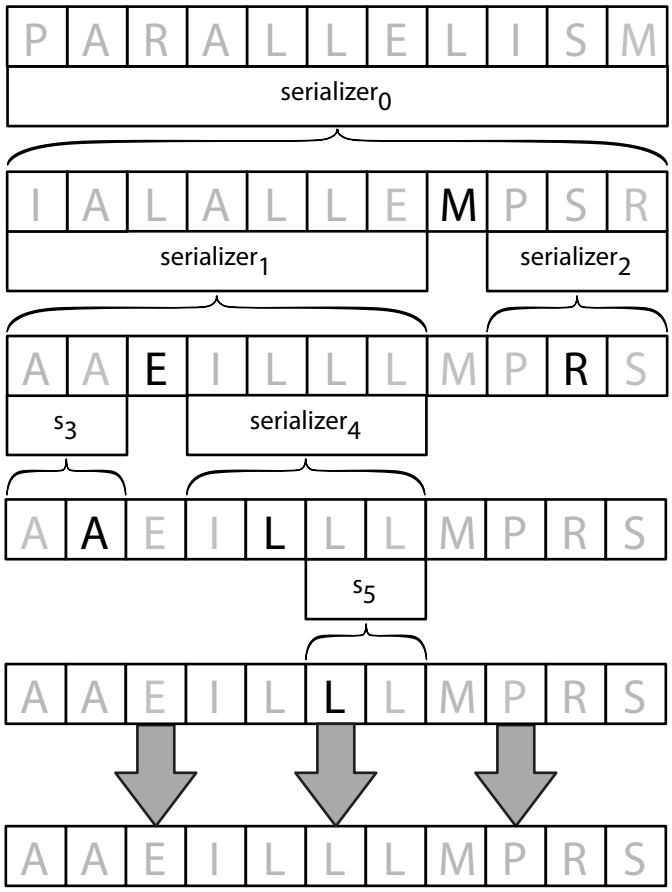


Figure 3.10: Recursive data subdivision in quicksort

ment in a block always serves as the pivot. Thus the `partition` function chooses "M" as the pivot element, and moves elements less than "M" to its left, and elements greater than or equal to "M" to its right. The `sort` method then creates two new `qsort_block_t` objects, which create new serializers when they are constructed. The `left` block owns the letters to the left of "M", and is associated with serializer$_1$. The `right` block owns the letters to the right of "M" and is associated with serializer$_2$. The `sort` method is then delegated to the serializer of these blocks.

The left and right blocks are partitioned in parallel. The right block, which contains "PSR", is partitioned around the pivot element "R", rearranging the letters to "PRS". Since the left and right sides of the pivot are both less than two characters,

```
1  void account_t::transfer (private_account_t* to_account, float amount)
2  {
3      this->withdraw (amount);
4      to_account->call (&account_t::deposit, amount);
5  }
```

Figure 3.11: Transfer method in PROMETHEUS

the recursion has completed for this side.

The left block is partitioned around "E", and then subdivided again into left and right blocks associated with serializer$_3$ and serializer$_4$, respectively. The `sort` method is then delegated to these serializers. The recursive process of dividing the data and sorting it continues until the sub-blocks are less than two characters and the recursion is complete. The result is the sorted array, shown at the bottom of Figure 3.10.

Nested delegation allows a programmer to apply data-driven decomposition to a data structure in a hierarchical fashion to reveal multiple layers of parallelism. In the next section, we present multiple delegation, which allows data-driven decomposition to expose multiple granularities of parallelism at a given layer in this hierarchy.

3.4 MULTIPLE DELEGATION FOR AGGREGATE OPERATIONS

The constructs we have described so far are sufficient to express potentially independent operations on individual objects, but client-side applications frequently contain *aggregate operations* that manipulate multiple objects at once. Consider the PROMETHEUS implementation of the `transfer` method, listed in Figure 3.11. This method operates on two account objects at once, withdrawing a specified amount from the receiver, and depositing this amount in the account specified by the first parameter (line 1). The `transfer` method does not meet our previous definition of object purity, so we cannot delegate this method invocation and instead must execute it sequentially. Sequential execution of aggregate operations inhibits parallelism, because later object-pure method invocations cannot be delegated until the transfer completes.

Consider the code listed in Figure 3.12. This snippet delegates object-pure method invocations to the serializers of `account1` and `account2` on lines 1–2. It then performs a transfer from `account1` to `account2` using the `call` interface

```
1  account1.delegate (&account_t::deposit, 50);
2  account2.delegate (&account_t::withdraw, 20);
3  account1.call (&account_t:transfer, account2, 100);
4  account2.delegate (&account_t::withdraw, 10);
5  account1.delegate (&account_t::withdraw, 20);
6  account3.delegate (&account_t::deposit, 40);
```

Figure 3.12: Invoking `transfer` via the `call` interface

(line 3). The `call` quiesces the serializer of `account1`, and once the previously delegated method invocations complete, sequential execution of the `transfer` method begins. Note that the transfer method also quiesces the serializer of `account2` (Figure 3.11, line 4). Because the transfer is executed sequentially, later object-pure method invocations on `account1` and `account2` (lines 4–5) cannot be delegated until the transfer completes. Worse yet, operations to accounts not involved in the transfer, such as the deposit to `account3` on line 6, must also wait for the transfer to complete.

We observe that this is an artificial limitation of the implementation we have described so far, rather than a fundamental limitation of data-driven decomposition. Recall that private objects are operated on by at most one method at any given time, and that these method invocations are applied in sequential program order. These invariants do not preclude the same operation from manipulating multiple objects. We therefore extend our notion of object purity to include methods whose side effects are confined to a set of objects, which we call *multi-object-pure* methods. Data-driven decomposition can exploit multi-object-pure methods by performing *multiple delegation*, which delegates the method invocation to the serializer of each object involved in the aggregate operation. To maintain the program ordering invariant, these method invocations are executed only once they are the oldest operation in every serializer to which they were delegated. Note that despite being delegated to multiple serializers, there is only one method invocation, so it is executed exactly once.

PROMETHEUS statically inspects the parameter list of delegated methods using template metaprogramming. If it finds any private objects, it specializes the delegation to perform the appropriate form of multiple delegation. This implementation of multiple delegation is similar to support for *multimethods* provided by some object-oriented languages, which allow a method to be specialized not only on the type of its receiver, but on the types of its arguments as well (Steele, 1990; Clifton

```
1 account1.delegate (&account_t::deposit, 50);
2 account2.delegate (&account_t::withdraw, 20);
3 account1.delegate (&account_t:transfer, account2, 100);
4 account2.delegate (&account_t::withdraw, 10);
5 account1.delegate (&account_t::withdraw, 20);
6 account3.delegate (&account_t::deposit, 40);
```

Figure 3.13: Aggregate delegation of the `transfer` method

et al., 2006).

While PROMETHEUS places no restriction on the number of private objects involved in multiple delegation, two considerations likely establish an effective upper limit. First, each private object must be passed as an argument to the method. Methods with many parameters tend to be cumbersome, so programmers typically do not write methods with more than six or seven parameters. Second, the overhead of delegation increases for each object involved in the aggregate operation. To date, we have found that by far the most common need for multiple delegation is interactions of two objects. Should we find it necessary to perform multiple delegation on large numbers of objects in the future, we could employ the techniques described in Chapter 6 for parallelizing delegation of large numbers of method invocations.

With support for multiple delegation, we can update the code from Figure 3.12 to delegate the transfer method invocation, as shown in Figure 3.13. Now we delegate the transfer on line 3 rather than execute it sequentially, and later operations on lines 4–6 need not wait for the transfer to complete before they are delegated. To demonstrate the operation of multiple delegation, we illustrate the execution of the delegated method invocations on `account1` and `account2` listed in Figure 3.13. We show their execution in six steps, spread across Figures 3.14 and 3.15.

Figure 3.14(a) shows the state of the program after delegation of the first two method invocations: a deposit to `account1`, and a withdrawal from `account2`. Figure 3.14(b) depicts delegation of the transfer method to the serializers of both accounts. In Figure 3.14(c), additional method invocations have been delegated to the serializer of each account.

In Figure 3.15(d), the serializer of `account1` has executed its first method invocation. Now the transfer is the oldest method invocation in this serializer, but it cannot execute it yet, since it is not the oldest method invocation in the serializer

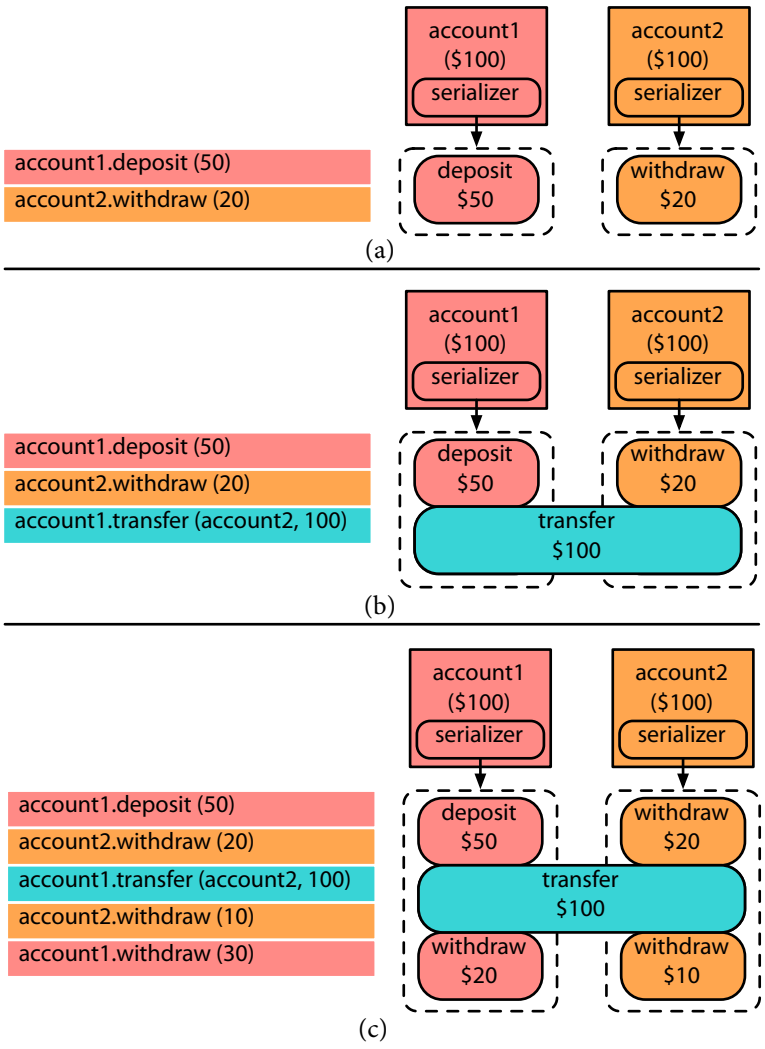


Figure 3.14: Multiple delegation of the `transfer` method

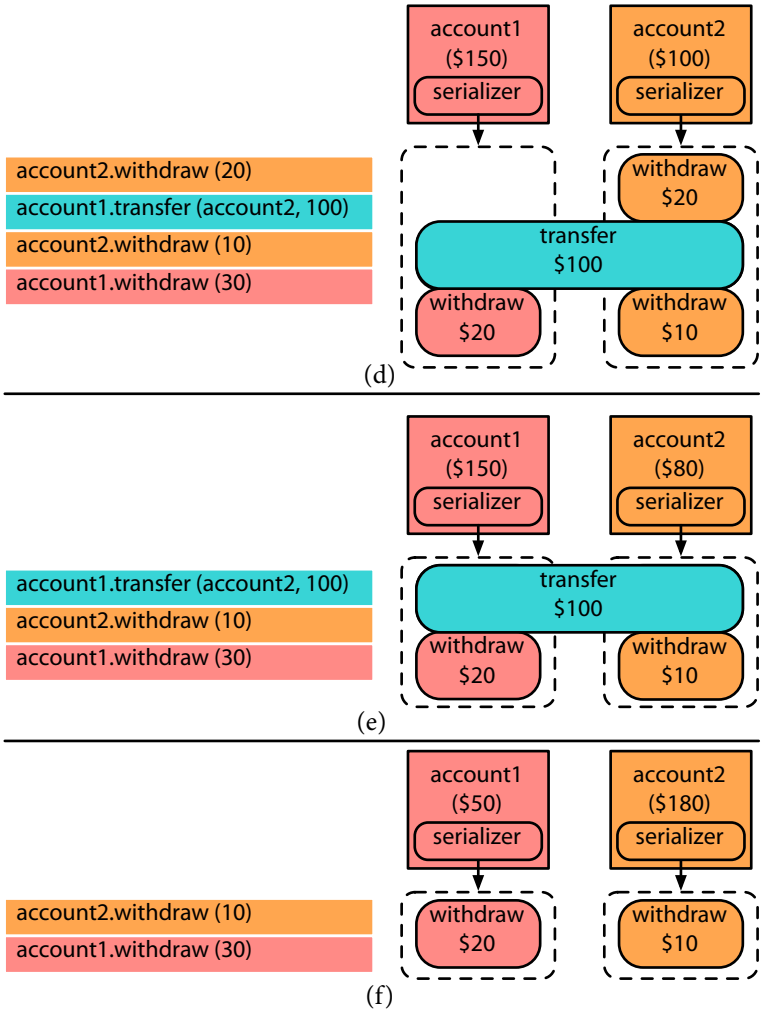


Figure 3.15: Multiple delegation of `transfer` method, continued

of `account2`. Only after the serializer associated with `account2` executes the withdrawal of $20, as shown in Figure 3.15(e), may the transfer method invocation execute. Figure 3.15(f) shows how once the transfer is complete, the serializers may resume independent execution of delegated method invocations.

This example illustrates how multiple delegation allows multi-object-pure methods to execute asynchronously, without inhibiting parallelism among later method invocations in the program. In concert with nested delegation, multiple delegation provides the ability to express parallelism at multiple different granularities of data throughout the program.

Multiple Delegation and Multimethods

PROMETHEUS statically inspects the parameter list of delegated methods using template metaprogramming. If it finds any private objects, it specializes delegation to perform the appropriate form of multiple delegation. Our implementation of multiple delegation is similar to the support for *multimethods* provided by some object-oriented languages (Steele, 1990; Clifton et al., 2006). Traditional object-oriented languages allow a method to be specialized on the type of the receiver object. Multimethods may be specialized not only on the type of the receiver, but also the types of arguments. An enhanced form of dynamic dispatch called *multiple dispatch* to is used to identify the correct method body for a multimethod based on the types of the receiver and the arguments (Chambers and Chen, 1999). With compiler support, similar techniques could be used to implement polymorphic delegation that automatically and dynamically performs the correct form of delegation for any method invocation.

3.5 CORRECTNESS AND SAFETY OF DATA-DRIVEN DECOMPOSITION

Implementing support for data-driven decomposition as a C++ library affords the significant advantages described in Section 3.2. However, this implementation also requires that we deal with the limitations imposed by an existing language. C++ is intended to be a flexible, general-purpose language supporting a wide range of programming styles, and not as a vehicle for strict object-oriented programming (Stroustrup, 1995). As a consequence of this design choice, the encapsulation guarantees of C++ objects are fairly weak. Furthermore, the permissive scoping and visibility rules of C++ preclude restricting a method's observable effects to its receiver object. (The stronger guarantees provided by type-safety in languages such as Java and C♯ prevents some, but not all, of the C++ limitations.) These

shortcomings make it impossible to enforce the exact semantics of private objects and object-pure methods. It is therefore possible to write a PROMETHEUS programs where a delegated method invocation accesses state that is simultaneously accessed by some other operation in the program, resulting in a determinacy race.

Though it does not realize the full potential of data-driven decomposition, PROMETHEUS represents a significant improvement in repeatability and predictability in comparison with control-driven decomposition. Determinacy races are an inherent property of shared-state multithreading, whereas they always represent errors in PROMETHEUS programs. Therefore we can rely on existing determinacy race tools such as those described by Feng and Leiserson (1997) and Bender et al. (2004) to detect these errors. We believe sacrificing safety for flexibility, backwards compatibility, and performance is a reasonable choice for PROMETHEUS and is consistent with the spirit of an unsafe systems language like C++.

In this section, we examine the cases where C++ allows violation of the assumptions of data-driven decomposition, possibly resulting in determinacy races. Rather than attempt to address every possible issue separately, we group these problems into broad categories that we illustrate with specific examples. We observe that nearly every source of inadvertently shared data is also a violation of the object-oriented practice of encapsulation. We show that using the private wrapper and a disciplined, object-oriented programming style can prevent almost all of these problems. At the end of this section, we examine how we might apply existing static analysis and type-checking techniques to implement a safe language for data-driven decomposition.

Violations of Private Object Semantics in C++

We begin by describing how C++ objects may violate the strong encapsulation requirements of private objects, illustrated with simple examples in Figure 3.16. We ascribe these issues to two categories: weaknesses of the C++ encapsulation mechanism, and the consequences of *aliasing* among the state encapsulated in different objects.

The first source of encapsulation violations in C++ programs is the encapsulation mechanism itself. C++ provides *class encapsulation*, meaning that the methods of an object may access the private members—both fields and methods—of any other object instantiated from the same class (Stroustrup, 1997, pg. 754). Figure 3.16 contains several examples of the consequences of class encapsulation. Consider the `read_others_field` method of the `example_t` class on lines 9–11. This method takes another object of the `example_t` class as a parameter, reads a

```
1  class example_t {
2      friend class other_class_t;
3
4  private:
5      int field;
6      int* pointer_field;
7
8  public:
9      int read_others_field (example_t& other) {
10         return other.field;
11     }
12
13     static int read_instance_field (example_t& object) {
14         return object.field;
15     }
16
17     int* get_field_address () {
18         return &field;
19     }
20
21     void set_pointer_field (int* pointer) {
22         pointer_field = pointer;
23     }
24 };
```

Figure 3.16: Violations of private object semantics in C++

field of that object (which is declared as `private` on line 5), and returns that value. Class encapsulation also allows class methods, which are declared in the class but not associated with an object instance, to access the private members of any object of that class. The `read_instance_field` method (lines 13–15), declared using the C++ `static` keyword (line 13), also reads a private field of an object of the `example_t` class. Both of these methods are legal in C++ and clearly violate the encapsulation assumption of private objects.

C++ also allows a class to declare *friends*—classes and functions that may then directly access the private fields of objects in this class. In Figure 3.16, the `friend` declaration on line 2 makes the private members of any object of the `example_t` class visible to any method of `other_class_t`. While certain practical circumstances necessitate the use of friends in C++ programs, it weakens encapsulation

and is subject to abuse.[5]

The private wrapper eliminates the problems of class encapsulation and friends by enforcing *object encapsulation*, which hides the private members of an object from every other object, regardless of class. The private wrapper hides the original class of an object using the protected inheritance feature of C++, so that other objects of the class, as well as objects or functions declared friends of the class, cannot access its private implementation. While the restrictions of object encapsulation may not be appropriate for all objects in C++ programs, we believe that reducing the probability of determinacy races justifies its use in the private wrapper.

The second source of encapsulation violations in C++ is *aliasing*. The fields of a C++ object may be pointers or references to state outside the boundary of the object. The state contained in the object is therefore the transitive closure of the data reachable from its fields. If this set of data is only reachable via the object, then it is *fully encapsulated* (Boyapati et al., 2003). In the parlance of OOP, the object *owns* this set of data. If any subset of the data reachable from an object's fields is reachable from outside of the object, then that subset is *aliased* (Hogg et al., 1992; Noble et al., 1998). Aliasing can cause a method to affect an object other than the receiver that is not obviously connected to the actions of that method. This is a problem in the context of OOP because it compromises the modularity of separate objects.

The code in Figure 3.16 shows two examples of how the fields of an object may become aliased. The first example is the `get_field_address` method (lines 17–19), which returns a pointer to `field`. The pointer returned from this method can be used to manipulate that field without using the object's methods. Leaking pointers into the internal representation of an object is usually viewed as poor object-oriented design, but some common programming idioms, including the *external iterator pattern* (Gamma et al., 1994, pg. 260), require it. The second example is the `set_pointer_field` method (lines 21–23), which assigns `pointer_field` to a pointer passed in as an argument. If the original pointer is maintained after calling this method, then it is an alias for `pointer_field`.

C++ makes no attempt to prevent aliases. However, it does provide unique pointers via the `unique_ptr` template, which allows the programmer to implement strict ownership semantics (ISO/IEC, 2010, sec. 20.9.10).[6] A unique pointer owns the data it points to, and automatically deallocates that memory when it goes

[5] The reputation of `friend` is evident in the C++ maxim "Friends don't let friends use `friend`!"

[6] The `auto_ptr` provided similar functionality in earlier versions of C++, but has been deprecated for the more robust `unique_ptr`.

out of scope. To enforce a single unique owner, these pointers cannot be copied, only moved, so that:

```
unique_ptr<int> old_p (new object_t); // Allocate an object
unique_ptr<int> new_p = old_p; // Compilation error!
```

is illegal, because it attempts to copy the `old_p` pointer, but:

```
unique_ptr<int> old_p (new object_t); // Allocate an object
unique_ptr<int> new_p = std::move (old_p); // Move pointer
```

works correctly, invalidating the `old_p` pointer and transferring ownership to the `new_p` pointer. To prevent aliasing in PROMETHEUS programs, programmers should prefer unique pointers to ordinary pointers for the fields of private objects.

Violations of Object-Purity in C++

C++ provides no mechanism to restrict the effects of a method to state owned by the receiver object, as is required for object-pure methods. A method may access external state both via global variables, or via its parameters. Figure 3.17 gives several simple examples of these problems.

The `write_global` method (lines 8–10) writes to `global_variable`, which is declared on line 1. PROMETHEUS cannot prevent methods from accessing global variables, because they are always visible in the scope of a method body. The detrimental effect of global variables on program understandability and maintainability is well-known (Wulf and Shaw, 1973). They are even more hazardous in object-oriented programming, because they violate modularity by creating a dependence on state external to the module that may be shared with a large number of other modules (Meyer, 1988a; Martin, 1996). Therefore programmers should follow standard OOP practice and avoid using global variables (especially non-constants) in PROMETHEUS programs.

Values passed by pointer or reference also allow a method to write to external state. Hogg et al. (1992) calls this *dynamic aliasing*, because the alias disappears when the method exits, in contrast to the *static aliasing* of fields, which persists across method invocations. The `write_argument` method (lines 12–14) gives an example of modifying external state by incrementing a value of a parameter passed via a pointer. PROMETHEUS is able to prevent these accesses to external state by statically rejecting delegation of a method that takes non-constant parameters by pointer or by reference.

The static checks performed by PROMETHEUS do not prevent a second class of

```
1  int global_variable;
2
3  class example_t {
4  private:
5      int field;
6
7  public:
8      void write_global () {
9          global_variable++;
10     }
11
12     void write_argument (int* argument) {
13         (*argument)++;
14     }
15
16     void read_constant (const int* argument) {
17         file = *argument;
18     }
19 };
20
21 void const_example () {
22     private_t <example_t> example;
23     int x = 0;
24
25     begin_delegation ();
26     example.delegate (&example_t::read_constant, &x);
27     x = 1;
28     end_delegation ();
29 }
```

Figure 3.17: Violations of object-purity in C++

errors that can result from parameters passed by pointer or reference. Consider the `read_constant` method (lines 16–18), which takes an integer value by `const` pointer, and assigns its value to `field`. The `const` qualifier prevents this method from writing to the pointed-to value, but it does not prevent external pointers to this value from modifying it. This problem is illustrated by the `const_example` function on lines 21–29. This function initializes the variable `x` to zero (line 23), then delegates the `read_constant` method (line 26), passing it the address of `x`. Next, `const_example` assigns the value of one to `x` (line 27), introducing a determinacy race. Because the execution of `read_constant` is asynchronous, it may read either zero or one when it dereferences the pointer to `x`.

To avoid creating this kind of race, programmers should ensure that values passed by pointer or reference to delegated method invocations are *relatively constant*, i.e., that their values do not change from the time the method is delegated to the next `end_delegation` call, which effects a local barrier. In our example, we could achieve this effect by moving the assignment to `x` on line 27 after the barrier on line 28. It would be straightforward to extend the PROMETHEUS interface with a wrapper template for arguments that could detect or prevent changes to variables passed by pointer or by reference, and require that the wrapper be used for arguments to delegated methods. We have elected not to require such a wrapper, as it would require wrapping every variable passed as an argument to a delegated method, and thus be much more invasive than our current annotations. Currently, we rely on programmer discipline to avoid this problem, and we continue to evaluate other possible solutions.

Towards a Safe Language for Data-Driven Decomposition

To realize its full potential for accessible parallel programming, data-driven decomposition should have language support that ensures its *safety*. Cardelli (2004) defines a safe language as one that prevents all *untrapped errors*—errors that can result in arbitrary behavior because they do not immediately cause an exception, such as accessing memory outside the bounds of an array. Determinacy races are also untrapped errors, so safe data-driven decomposition must prevent these errors using some combination of static and dynamic checking. To achieve safety, we could utilize existing static analysis and type-checking techniques to enforce strong encapsulation of private objects and verify that the effects of delegated methods meet the requirements for object purity.

There are two existing approaches to enforcing strong encapsulation. The first approach is to enforce restrictive coding conventions with static checking

(Hogg, 1991; Almeida, 1997). These techniques could be employed to enforce our recommended programming discipline. The second approach is to augment a language with support for *ownership types* (Clarke and Drossopoulou, 2002; Boyapati et al., 2003), which allow programmers to express ownership relationships between objects and statically detect violations of these relationships. Ownership types aim to provide greater flexibility then the restrictive approach, so that idioms like external iterators may be used without violating encapsulation. If private objects were enhanced with ownership types, we might be able to imbue them with less restrictive semantics while still satisfying the necessary conditions for data-driven decomposition.

Type-and-effect systems (Lucassen and Gifford, 1988) extend conventional type systems to express the *effects* of a computation in terms of reads or writes to abstract regions of memory. For example, the effects of an object-pure method would be expressed as writes to the region containing the receiver object, and reads of the regions of the method's parameters. Recent work by Bocchino et al. (2009b) for checking the effects of Java methods could be adapted to check that delegated methods meet the requirements of object purity. Effect inference techniques, such as those proposed by Benton and Fischer (2009) and Vakilian et al. (2009) might be applied to automatically identify object-pure methods suitable for delegation.

Given the recent progress in these areas, we are optimistic that language support for safe data-driven decomposition is an achievable goal. We leave further exploration of this issue for future work.

3.6 RELATED WORK

Many other parallel execution models have been proposed to combat the difficulties associated with nondeterminate parallel execution. These models are described as deterministic or determinate, but these terms are not used consistently in the literature, so we will classify each according to our taxonomy from Chapter 2: *sequentially determinate* (predictable) models that guarantee the sequential ordering of assignments to each variable when the program is run with a given input, and *determinate* (repeatable) models that guarantee the same (but not necessarily sequential) ordering of assignments to each variable when the program is run with a given input. We begin by examining several models that provide sequential determinacy.

Jade (Rinard and Lam, 1998) extends the C language with support for implicit, dynamic parallel execution. Jade provides the abstraction of *shared objects* which may be accessed by any code in the program. Programmers facilitate dynamic parallelization in Jade by indicating blocks of code that are candidates for parallel execution, which are called *tasks*. The programmer also annotates each task with a set of *access specifications* that specify the shared objects the task will access, and whether the task will read, or read and write the object. Access specifications can contain expressions that are evaluated dynamically to determine the particular objects that will be accessed by the task. When a Jade program is executed, the runtime system orchestrates execution of the tasks by evaluating the access specifications of each task it encounters, and using the results to schedule the task for execution according to its dependences on the access specifications of other tasks. As the program runs, Jade dynamically checks that each task is faithful to its access specification.

Jade and PROMETHEUS share the common goal of efficiently executing sequentially determinate programs, and they both achieve this goal by dynamically decomposing the program so that they may identify the data accessed by each operation. However, the abstractions they provide are very different. Jade provides shared objects that may accessed by any operation, while PROMETHEUS provides private objects that may only be manipulated their methods. The trade-off for the programming model is that Jade's access specifications make it easier to express operations on arbitrary sets of data, whereas PROMETHEUS is more restrictive; but Jade makes significant changes to the sequential programming model, while PROMETHEUS is able to present a model that closely resembles standard sequential OOP. These differences also have implications for the runtime system: Jade uses a custom runtime system to meet the requirements of its model; in Chapter 4, we will show that PROMETHEUS is able to exploit support for dynamic scheduling present in many libraries and languages for parallel programming.

Duran et al. (2009) propose StarSs, an extension to the OpenMP 3.0 task model. OpenMP tasks use a similar programming interface to the tasks of Cilk, which we describe in detail in Chapters 4 and 5. StarSs extends this task programming model to support *dependent tasks*, where each task is augmented with a set of *directionality clauses* that specify that a particular set of memory locations are used as input, output, or both input and output for the task. These clauses are used by the StarSs runtime system to schedule tasks according to their dynamic data dependences in a similar fashion to Jade's access specifications. The runtime system

automatically avoids false dependences (write-after-read and write-after-write) by renaming the memory locations used by tasks with false dependences.

Bocchino et al. (2009b) propose Deterministic Parallel Java (DPJ), a dialect of the Java language augmented with parallel constructs and a type-and-effect system to statically guarantee the safety of parallel execution. DPJ provides `cobegin` and `foreach` constructs for expressing parallelism in the program. Programmers also annotate the program to associate data with abstract *regions* of the heap, and to indicate a summary of the *effects*—either read or write—of each method on the regions it accesses. The compiler uses the region and effect information to verify that each parallel operation does not write data that is read or written by other parallel operations, or that synchronization ensures a correct ordering. The parallel program is then guaranteed to produce the desired ordering.

The type-and-effect system of DPJ provides strong safety guarantees, which are clearly a desirable property of any parallel execution model. However, DPJ is only able to exploit parallelism that can be statically verified, which would limit its application to irregular programs with unstructured parallelism. Pairing the powerful safety guarantees of DPJ's type-and-effect system with an ordering mechanism like the serializers proposed in the next chapter could potentially provide the benefits of safety and dynamic decomposition.

Grace (Berger et al., 2009) supports the fork-join task parallel programming model with a software runtime system that ensures the sequential determinacy of the resulting execution. Grace runs tasks in different processes, and exploits virtual memory protection to isolate the memory references of each task. Memory updates are integrated using a sequential commit protocol that only writes back the results of a task after all previously ordered tasks have committed, and it has verified that the values read by the task have not changed, indicating a violation. Grace's novel use of memory protection hardware might be leveraged to ensure the dynamic safety of data-driven decomposition.

Determinate Parallel Execution Models

Determinate program ordering do not have to be derived from the program itself—instead, they can be imposed dynamically. In contrast with data-driven decomposition, these models do not provide the advantage of a simple, predictable programming model to the programmer. However, because they do not change the programming model, they are directly applicable to existing programs, such as legacy multithreaded code.

Devietti et al. (2009) propose deterministic shared memory multiprocessing (DMP), and evaluate several hardware-based schemes that enforce a determinate ordering on memory accesses. They begin with a design that serializes the memory accesses of all processors, resulting in a truly deterministic ordering. Devietti et al. show that this scheme incurs unacceptable overhead, as would be expected. They then propose a set of mechanisms to exploit hardware support for transactional memory to relax the deterministic ordering to a determinate ordering, and show that these techniques can greatly reduce the overheads of enforcing determinacy.

In follow-on work to DMP, Bergan et al. (2010) propose CoreDet, a compiler and runtime environment for writing determinate parallel programs. CoreDet divides dynamic program execution into a series of quanta, where each quantum has two phases: a parallel mode for known-independent memory accesses, and a serial mode for potentially dependent memory accesses. Bergan et al. describe two different approaches that employ these quanta to ensure determinacy. The first approach employs a table to track the identity of the thread that owns each memory location. A thread may freely access any data it owns during parallel mode, but must wait until serial mode to access other data, at which point it may take ownership of that data. The second approach buffers the stores of each thread during the parallel phase of a quantum, and commits them serially at the end of the parallel phase. Because a thread may read its store before the end of the parallel phase, the resulting program is not sequentially consistent. In their evaluation, Bergan et al. (2010) show that while the ownership table has lower overhead, but the store buffers are more scalable.

Kendo (**?**) is a software-only system that enforces a repeatable ordering on all lock acquisitions when a program is run with a particular input. Kendo uses a notion of *deterministic logical time*, where each thread counts the frequency of some repeatable event, and use this counter to determine ordering of a thread with respect to other threads. It also uses a modified implementation of pthread locks that check logical clocks and only allow a thread to acquire a lock when it is its turn. If a Kendo program correctly quiesces all of its accesses to shared data, then it is determinate.

Synchronizing Data Manipulation via Serialization

Serialization has been used by a number parallel execution models as a mechanism to provide atomicity of data manipulation. Two notable examples are the Actor model and Apple's Grand Central Dispatch.

The Actor model (Hewitt and Steiger, 1973) unifies the notions of data objects

and threads of execution. Each actor conceptually executes in its own thread, and actors communicate with each other via asynchronous message passing. These messages are processed serially by each actor, typically in the order of their arrival. The Actor model was later codified as the *active object* pattern (Lavender and Schmidt, 1995) for object-oriented programming.

Apple recently introduced Grand Central Dispatch (GCD) (Apple, 2009), which abstracts parallel computation via a set of different types of *dispatch queues*. Parallel work is expressed via *blocks*, an extension to the C, C++, and ObjectiveC languages that denote a region of static code as a unit of parallel work. Block instances are scheduled for execution by the operating system using a set of dispatch queues synchronized with atomic operations. *Private dispatch queues* are one of the queue types provided by GCD that may be used by programmers to ensure atomic access to shared data structures. If block instances are submitted to a private dispatch queue in program order, then their execution will be determinate. Unlike serializers, dispatch queues must be manually managed by programmers.

3.7 SUMMARY

In this chapter, we proposed data-driven decomposition, a parallel execution model that dynamically partitions a sequential program into units of parallel work based on the data accessed by the operations of the program. We introduced the private object and serializer constructs and described how they are used to perform data-driven decomposition. Private objects are fully encapsulated, disjoint sets of data that may safely be manipulated in parallel. Each private object is associated with a serializer that asynchronously executes method invocations one-at-a-time, in program order. These constructs are sufficient to ensure sequential determinacy, so the resulting parallel execution is repeatable and predictable.

We also described the programming interface of the PROMETHEUS C++ library. Using PROMETHEUS, programmers express data-driven decomposition using the familiar constructs of object-oriented programming. We then showed how nested delegation and multiple delegation allow the programmer to flexibly express potential independence at different granularities throughout the program. We concluded the chapter by showing that a disciplined, idiomatic application of object-oriented programming minimizes the likelihood of errors in PROMETHEUS programs, and speculated how we might employ existing programming language techniques to ensure the safety of data-driven decomposition.

4 SERIALIZER DESIGN AND IMPLEMENTATION

> *A corollary of the determinacy theorem is that the entire sequence of values written into each and every memory cell during any run of the system is the same for the given input. This corollary also tells us that any system of blocking tasks that communicates by messages using* FIFO *queues (instead of shared memory) is automatically determinate because the message queues always present the data items in the same order to the tasks receiving them.*
>
> — Peter J. Denning and Jack B. Dennis (2010)

In Chapter 3, we introduced the private object and serializer constructs for performing data-driven decomposition. During the presentation of the execution model and the PROMETHEUS API, we discussed private objects in detail, but so far we have only addressed the serializer as a programming abstraction. In this chapter we begin by discussing the necessary components of any serializer design. We then describe the design and implementation of the PROMETHEUS serializer, and show how the `delegate` and `quiesce` operations manipulate this construct to realize data-driven decomposition. Our serializer design utilizes the fork-join task programming capabilities provided by many current languages and libraries.[1] These platforms use dynamic scheduling to parallelize task execution, under the assumption these tasks are independent. Using a novel technique called *dynamic task extension*, our design is able to exploit the efficient scheduling provided by these systems, while still enforcing the dynamic dependences between method invocations resulting from a data-driven decomposition.

4.1 SERIALIZER DESIGN CONSIDERATIONS

A serializer has two primary responsibilities: First, it must maintain the invariant that the method invocations it is delegated are executed one-at-a-time, in program order. Second, it must endeavor to parallelize the serialized execution of these method invocations with the rest of the program. To fulfill these responsibilities, any serializer design must have a data structure to track method invocations and a mechanism to schedule these method invocations for execution.

[1] To avoid confusion with other uses of the term *task*, in this dissertation we use use the term *task* exclusively to refer to the lightweight threads of the dynamically scheduled fork-join execution model popularized by Cilk (Frigo et al., 1998).

The Serialization Queue

To preserve the serialization invariant, the serializer must incorporate a data structure to track method invocations that maintains their ordering. We call this structure the *serialization queue*, because a queue is the abstract data type (ADT) that best meets this requirement. Note that the implementation of this abstract data type will depend on the specific goals of the serializer design, the scheduling mechanism, and the target software and hardware platforms.

The queue ADT defines three operations: `produce`, which inserts an item into the queue; `consume`, which removes an item from the queue in a first-in first-out (FIFO) manner; and `empty`, which is used to determine if they queue contains any items. These operations provide a sufficient basis for implementing a serializer. The `produce` operation is used by delegation to insert method invocations into the queue. The `empty` operation is queried by synchronization to check if all method invocations delegated to the serializer have completed. Runtime scheduling operations also query `empty` to check if the serialization queue contains method invocations, and uses `consume` to remove an invocation to execute.

Serializer Scheduling

In addition to tracking method invocations and their ordering, the serializer must *schedule* execution of these invocations by assigning them to a *worker thread*—an abstraction of a hardware execution context that is managed by the runtime. The primary goal of scheduling is to guarantee the serialization of method invocations for a particular serializer. The secondary goal of scheduling is to parallelize execution of method invocations from different serializers to the greatest degree possible.

Scheduling may be either static or dynamic. It may seem counter-intuitive that a dynamic decomposition could employ static scheduling—to clarify the distinction, we define scheduling as static if a fixed formula for assigning method invocations to worker threads is encoded in the program, so that when a program is run with a particular input and a certain number of worker threads, a given method invocation is always assigned to the same worker thread. Scheduling is dynamic if the assignment of method invocations to worker threads may change in different executions with the same input and number of worker threads.

Serializer Design Goals

Before describing the serializer implementation in PROMETHEUS, we identify several desirable properties of a serializer. We focus on useful characteristics for data-driven decomposition on general-purpose multiprocessor and multi-core systems, such as those used in servers, desktop and laptop computers, and mobile devices. A serializer for more specialized platforms, such as graphics processors (Owens et al., 2007; **?**) and heterogeneous systems like the IBM Cell processor (Gscwind, 2007), might have a different set of goals that reflect the needs of the specific kinds of applications targeting these platforms.

Dynamic Scheduling. A serializer for general purpose parallel computing should be amenable to dynamic scheduling, which relies on a runtime system to assign operations to worker threads for execution. This relieves programmers of the scheduling burden, allowing them to focus on the expression of parallelism in a program. It also allows the program to tailor scheduling to the particulars of the hardware on which it is running, such as the number of cores in a multi-core processor. Dynamic scheduling performs load balancing, which improves overall performance when parallel computations perform varying amounts of work (Mohr et al., 1991). Load balancing is especially useful for multiprogrammed systems, where the operating system may preempt worker threads in favor of unrelated processes, so that some worker threads to appear to run more slowly than others (Arora et al., 1998).

Locality Optimization. The method invocations of a serializer exhibit both spatial and temporal locality, because they manipulate a single receiver object, or a small set of objects. Serializer scheduling should exploit this locality by successively executing all method invocations in a particular serializer until it is empty, rather than alternately executing method invocations from different serializers.

Concurrent Delegation and Execution. A serializer should allow delegation of method invocations at the same time the runtime is executing earlier method invocations from the same serializer. If delegation to an executing serializer blocked, it would inhibit the discovery of further parallelism in the program. The key to satisfying this goal is to ensure that the serialization queue is a concurrent structure that allows simultaneous produce and consume operations.

Dynamically Sized Serialization Queue. Implementing a serializer with a fixed-size serialization queue introduces several problems. If it is too small, it may fill up and cause delegation to block. If it is too large, it wastes memory and hampers locality. In general, it is not possible to identify an upper bound on the number of method invocations that may be in a serialization queue during program execution, since this will depend on the input to the program, as well as the relative rates of delegation and execution of these method invocations in a particular run of the program. Therefore the serialization queue should not be of a fixed size, but rather should be capable of growing and shrinking to accommodate the number of method invocations contained in the serializer at any given time.

Nonblocking Operations. We can generalize the previous two goals by guaranteeing a *progress condition* for the serialization queue. Progress conditions characterize the effect that a thread experiencing a delay—which may be caused by a cache or TLB miss, a page fault, or preemption—has on other threads accessing the same concurrent data structure (Herlihy and Shavit, 2008, pg. 59–61). If the delay of one thread prevents other threads accessing the structure from making progress, we say the data structure is *blocking*. If the delay of one thread does not necessarily delay other threads, then the data structure is *nonblocking*.[2] To ensure good performance, our serializer design should endeavor to guarantee the strongest possible progress condition for the serialization queue (Moir and Shavit, 2004).

With these goals in mind, we proceed to describe the design of the PROMETHEUS serializer.

4.2 A TASK-BASED SERIALIZER DESIGN

The fork-join task programming model has been widely adopted by libraries and languages for parallel programming. A *task* is essentially a very lightweight thread of control—the overhead of spawning a task is typically a few times the overhead of an ordinary function call (Frigo et al., 1998). In contrast with the high overhead of spawning a thread, the low cost of creating tasks allows the programmer to apply control-driven decomposition to very fine-grained operations, and relieves

[2] Some authors use the term *nonblocking* as a synonym for the *lock-free* progress condition (see Section 4.6). In this dissertation we use the broader definition of nonblocking given by Moir and Shavit (2004) and Herlihy and Shavit (2008).

```
1  class task_t {
2  public:
3      // Pure virtual method implemented by subclasses of task_t
4      virtual void execute () = 0;
5  };
```

Figure 4.1: Task interface

the programmer of the responsibility of assigning these tasks to threads. Runtime support dynamically schedules tasks for execution, typically using efficient algorithms with provable time and space bounds (Blumofe and Leiserson, 1999). Dynamically scheduled task programming was popularized by Cilk (Frigo et al., 1998), and is supported in many other parallel platforms, including OpenMP 3.0 (OpenMP, 2008), Intel's Threading Building Blocks (TBB) (Reinders, 2007) and Cilk++ (Leiserson, 2009), the Java Fork/Join framework (Lea, 2000), and Microsoft's .NET Task Parallel Library (TPL) (Leijen et al., 2009). Task programming is also supported in many recent languages, including those developed for the DARPA HPCS program: Chapel (Chamberlain et al., 2007), Fortress (Allen et al., 2007), and X10 (Charles et al., 2005).

If we implement serializer scheduling using task programming primitives, we can leverage the efficient scheduling already provided by these platforms. This presents a significant challenge, because dynamic scheduling algorithms assume tasks are independent, whereas the method invocations produced by a data-driven decomposition may depend on earlier method invocations. In this chapter, we present a serializer design that overcomes this challenge using a technique called *dynamic task extension* to ensure the serial execution of method invocations in a given serializer, while exploiting existing task scheduling support to execute method invocations in different serializers. Using tasks as the basis for scheduling serializers does not mean that the task programming interface must be exposed to programmers for them to use data-driven decomposition—these operations may be entirely hidden in the runtime system. Before describing the design and implementation of the PROMETHEUS serializer, we review the basics of task programming, which will clarify the expression of our serializer in terms of tasking operations.

```
1  class deposit_task_t : public task_t {
2  private:
3      account_t* account;
4      float amount;
5
6  public:
7      // Constructor
8      deposit_task_t (account_t* account, float amount) :
9          account (account), amount (amount)
10     {}
11
12     // Implementation of execute method
13     virtual void execute () {
14         account->lock ();
15         account->deposit (amount);
16         account->unlock ();
17     }
18 };
```

Figure 4.2: Deposit task

Task Programming Review

We present our task programming examples using a C++ interface similar to that of TBB. Tasks are implemented as classes, and objects of these classes are instantiated to create dynamic tasks as the program executes. Figure 4.1 lists the interface for the `task_t` class which has a pure virtual method `execute` (line 4). To create a new kind of static task, the programmer specifies a class inheriting from `task_t` and implements the `execute` method to perform the desired operations. The programmer may also add fields to the object, which are used to communicate values the task will need when it executes.

The runtime assumes that tasks are independent, i.e., that they do not write to memory that other tasks read or write. Therefore the runtime may schedule tasks concurrently, and interleave them in any way. As with any control-driven decomposition, if a task accesses shared state, the programmer must use critical sections, or some other form of synchronization, to ensure atomicity of the operation.

Figure 4.2 provides a task class for performing a deposit, suitable for use in the bank transaction processing program introduced in Chapter 2. The `deposit_task_t` class inherits from the `task_t` class (line 1). It adds fields

to track the account the task will manipulate (line 3) and the amount to deposit (line 4). The constructor (lines 8–10) initializes these fields to values specified by the constructor's parameters. The `deposit_task_t` class implements the `execute` method (lines 13–17) to call `deposit` on the appropriate account. Since this account may be accessed by other concurrently executing tasks, the deposit is enclosed in a critical section to prevent other tasks from simultaneously accessing the same account.

To submit a task for execution, the programmer must first instantiate an object of the desired task class, and then call `spawn` to submit it for execution. Runtime support dynamically schedules tasks and ensures that each spawned task is executed exactly once (Michael et al., 2009). The following code shows how the programmer might create and spawn an instance of the deposit task:

```
task_t task = new deposit_task_t (account, amount);
spawn (task);
```

Besides `spawn`, the other primitive operation provided by task programming is `sync`, which performs a local barrier synchronization that waits until all outstanding tasks have completed. Task programming allows both `spawn` and `sync` to be used inside of tasks to express nested parallelism.

Dynamic Task Extension

Delegating a method invocation looks superficially similar to spawning a task, so one might be tempted to implement delegation using a task for each method invocation. However, tasks are ill-suited to execute these method invocations. Each delegated method invocation in a data-driven decomposition may be independent, or may depend on earlier method invocation, while tasks are scheduled under the assumption that they are independent, and no ordering is imposed on their execution. Recognizing that tasks are only appropriate for independent computations, we can instead utilize them to schedule different serializers, so that an individual task is responsible for all of the method invocations delegated to a serializer.

Using tasks to implement serializer scheduling is not as simple as just creating a task for a serializer when it is delegated a method invocation. Conventional tasks execute a single invocation of a function or method in the program, while a serializer may comprise multiple method invocations that are dynamically identified during program execution. So while a task executes a method invocation from a serializer, additional invocations may be delegated to that same serializer.

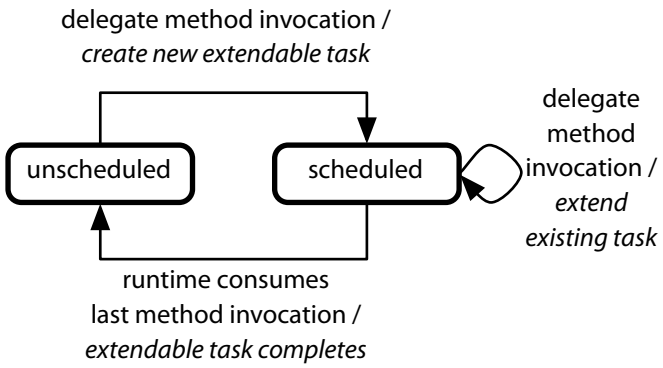


Figure 4.3: State diagram for serializer scheduling

We have already established that there should be a single task for each serializer, so in this case, we should not create an additional task. We also do not want to require delegations to wait for a running task to complete, as that would inhibit the discovery of parallelism later in the program—one of the goals we established for our serializer design in Section 4.1 was that it should allow concurrent delegation and execution of method invocations.

We rectify the shortcomings of conventional tasks for scheduling serializers by augmenting the existing task class to create a class of *extendable tasks*. An extendable task executes a series of method invocations, rather than a single method invocation, and supports *dynamic task extension*, so that additional method invocations may be appended to the task while it is running. We then employ an extendable task to schedule all of the method invocations delegated to a given serializer. If additional method invocations are delegated to a serializer while this task is executing, they extend the existing task, rather than triggering the creation of a new task.

Figure 4.3 depicts a state transition diagram for serializer scheduling via dynamic task extension. A serializer is always in one of two states: it is either *scheduled*, or *unscheduled*. These states reflect whether or not an extendable task exists for the serializer, and transitions between these states occur during delegation and task completion. In the figure, the standard text on transition arcs indicates input actions that cause a state transition, while the italicized text indicates the transition action.

When a serializer is created, its serialization queue is empty, and it is in the *unscheduled* state. The first `delegate` operation to this serializer creates a new

```
1  struct serialization_node_t {
2      serialization_node_t* volatile next;
3      invocation_t* invocation;
4
5      // Constructor
6      serialization_node_t (invocation_t* invocation) :
7          next (NULL), invocation (invocation)
8      {}
9  };
```

Figure 4.4: Serialization queue node structure

method invocation record and inserts it into the serialization queue, and transitions the serializer to the *scheduled* state. Then the `delegate` operation creates a new extendable task for the serializer and spawns that task for execution by the runtime. If a future method invocation is delegated to this serializer while it is still in the scheduled state, indicating there is already a task executing its method invocations, then delegation extends the existing task with the new method invocation. When an extendable task finishes executing the last method invocation in the serialization queue, the task completes, and the serializer transitions to the unscheduled state. If another method invocation is delegated to this serializer at some future point in the program's execution, then a new extendable task must be created to schedule its execution.

Dynamic task extension provides a mechanism for scheduling serializers using existing support for task scheduling, while ensuring that there is never more than one task executing the invocations of a particular serializer. With this design in mind, we now proceed to detail the implementation of the PROMETHEUS serializer, beginning with the serialization queue.

4.3 THE SERIALIZATION QUEUE

To meet the design goals outlined in Section 4.1, the serialization queue should be a dynamically sized data structure that allows concurrent `produce` and `consume` operations. The single-producer single-consumer queue described by Lamport (1983) uses an array for data storage and synchronizes accesses with ordinary loads and stores.[3] Because this queue is synchronized without resorting to mutual

[3] Lamport provides the earliest known description of this queue, but according to Herlihy and Shavit (2008, pg. 65), this design is considered folklore.

exclusion, it provides extremely efficient concurrent access. However, this structure is unsuitable for our design, because it uses a fixed-size array. Hendler and Shavit (2002) present a producer-consumer buffer (PCB) with a similar design that replaces the array with a linked list structure. The size of the linked list is inherently dynamic, reflecting the number of items in the queue. Thus the Hendler-Shavit PCB satisfies the goals of our design, and we use it as the basis of our serialization queue.

Figure 4.4 lists the code for an individual list node for the serialization queue. Each node maintains a pointer to the next node in the list (line 2) and a pointer to the data payload—in this case a method invocation object (line 3). Each of these method invocation objects stores a pointer to the receiver of the method, a pointer to the method to be invoked, and the arguments to be passed to the method. (We do not show the implementation of the method invocation class, which is conceptually simple but requires a large amount of cryptic template code.) The constructor for the list node (lines 6–8) accepts a pointer to an invocation object, and initializes the node's invocation pointer to this value.

Figure 4.5 shows the serialization queue implemented as a Hendler-Shavit PCB. The fields of the class are pointers to the head node (line 3) and the tail node (line 4). The constructor sets both the head and tail to point at a newly created dummy node, which is initialized with both its `next` and `invocation` pointers set to `NULL`. This dummy node serves as the first *sentinel node*, a node with no data that is always at the head of the list. The serialization queue is empty if and only if the next pointer of the sentinel node is `NULL`, as shown in the `empty` method (lines 12–15). Note that when the list is empty, the `tail` pointer will also point to the sentinel node.

The `produce` method (lines 17–22) takes a pointer to an invocation object as its single parameter. It creates a new node to hold the invocation, and sets the `next` pointer of the node pointed to by `tail` to point at the new node (line 20). This is the *linearization point* of the `produce` method: the single step where the effects of the method become visible to concurrent operations on the PCB (Herlihy and Wing, 1990). In particular, this is the step where the new invocation becomes visible to invocations of `consume`. The last action of the method is to set the `tail` to point to the new invocation. The use of the sentinel node greatly simplifies the `produce` method, because its presence at the head of the list ensures that `tail` always points to some list node, allowing the cases where the list is empty and non-empty to be treated the same way.

The `consume` method (lines 24–29) saves the node currently at the head of the list in the local variable `old_head`. It then sets `head` to point at the next node in

```
1  class serialization_queue_t {
2  private:
3      serialization_node_t* head;
4      serialization_node_t* tail;
5
6  public:
7      // Constructor
8      serialization_queue_t () :
9          head (new serialization_node_t (NULL)), tail (head)
10     {}
11
12     bool empty () {
13         if (head->next == NULL) return true;
14         else return false;
15     }
16
17     void produce (invocation_t* invocation) {
18         serialization_node_t* node;
19         node = new serialization_node_t (invocation);
20         tail->next = node;
21         tail = node;
22     }
23
24     invocation_t* consume () {
25         serialization_node_t* old_head = head;
26         head = head->next;
27         delete (old_head);
28         return head->invocation;
29     }
30 };
```

Figure 4.5: Serialization queue class

the list (line 26). This assignment is the linearization point of the `consume` method, and is the single step that may cause the serialization queue to transition from the non-empty state to the empty state. Next, the `old_head` node is deallocated, and the invocation in the current head of the list is returned. The list node that previously held this invocation now becomes the new sentinel node for the list. Note that the `consume` method assumes the queue contains at least one entry, so the `empty` method must always be queried to ensure the queue is not empty before calling `consume`.

Like the single-producer single-consumer queue described by Lamport, the Hendler-Shavit PCB employs no locks, and is synchronized entirely with ordinary loads and stores. The key to the operation of the Hendler-Shavit PCB is the sentinel node which is either the initial dummy node, or the list node that held the last invocation object returned from `consume`. Representing the empty queue with both `head` and `tail` pointing to the sentinel node, rather than being set to `NULL`, allows the cases of empty and non-empty queues to be treated the same way: the `consume` method may remove the last item in the buffer without changing the `tail` pointer, and the `produce` method to add to an empty buffer without changing the `head` pointer. This allows the `produce` and `consume` methods to be linearized by changing the `next` pointer of the head and tail nodes, respectively, so they are completely decoupled.

Despite its desirable properties, the Hendler-Shavit PCB is not sufficient to implement a serializer. Using this structure without modification causes a race condition when an extendable task is consuming the last method invocation in the serialization queue while a concurrent `delegate` operation is producing an additional method invocation into the queue. Consider the following case where the serializer is in the scheduled state, and the serialization queue contains one or more method invocations: Before producing an invocation, the `delegate` operation sees that the queue is not empty, and thus the serializer must be scheduled. Next, the task serializer's task consumes and executes all the method invocations from the serialization queue. Seeing that the queue is empty, this task ceases execution and the serializer transitions to the unscheduled state. Finally, the `delegate` operation adds its method invocation, but does not create a new task and spawn it for execution because it assumes that the invocation extended a currently executing task. In this scenario, the newly delegated method invocation may never be executed, because the `delegate` operation and the extendable task both wrongly assume the other is responsible for scheduling the method invocation.

While the `produce` and `consume` operations of the Hendler-Shavit PCB are completely decoupled, the serializer requires that the producer and consumer

coordinate to determine whether the queue ever becomes empty. If it does, then the `delegate` operation must create a new task to execute the serializer. If the queue does not ever become empty, the current task must assume responsibility for the additional method invocation. In the next section, we will describe how to coordinate concurrent `produce` and `consume` operations so they agree on the status of the serialization queue.

4.4 SCHEDULING SERIALIZERS

To enable concurrent `produce` and `consume` operations on the serialization queue to agree on whether the queue ever becomes empty, we augment the serializer with a tag. We define a tag value of zero to indicate an empty queue, and positive integers to indicate that the queue is non-empty. This tag is updated by the `produce` and `consume` methods using the atomic compare-and-swap (CAS) primitive to ensure that concurrent operations achieve consensus on the state of the queue.

The code for the serializer class is shown in Figure 4.6. The fields of this class include the serialization queue (line 5) and a 64-bit unsigned integer used to implement the tag (line 6). The tag is initialized to zero in the constructor (lines 9–11), and may be modified using the provided accessor (lines 13–14) and mutator (lines 16–18) methods.

The serializer class also provides the `cas_tag` method (lines 22–25) that takes the presumed value of the tag and a new value as parameters. This method updates the tag using the compare-and-swap (CAS) instruction provided by many processor implementations (IBM, 1975; SPARC, 1994; Intel, 2009a).[4] CAS performs the following actions atomically: first, it reads the current value of a specified memory location; second, if this value is equal to the presumed value, it stores the new value to that memory location. The CAS instruction returns the previous value of the memory location, which indicates if the swap was successful. If the previous value is equal to the presumed value, then the new value was stored to the memory location; otherwise, it was not. The serializer also has `produce` (line 28) and `consume` (line 31) methods that are used to implement delegation and the serializer task, respectively.

The `produce` method, listed in Figure 4.7, takes a method invocation object as a parameter. It returns `true` if the serializer task should be created and spawned, and

[4] If the processor's instruction set architecture does not implement CAS, other atomic primitives such as load-linked/store-conditional (LL/SC) may be used (Compaq, 2002; IBM, 2005; MIPS, 2008).

```
1  typedef uint64_t tag_t; // Use 64-bit unsigned integer
2
3  class serializer_t {
4  private:
5      serialization_queue_t queue;
6      tag_t tag;
7
8  public:
9      serializer_t () :
10         tag (0)
11     {}
12
13     tag_t get_tag () {
14       return tag; }
15
16     void set_tag (tag_t value) {
17         tag = value;
18     }
19
20     // Update tag field with compare-and-swap instruction
21     // Returns true if update is successful, false if not
22     bool cas_tag (tag_t old_value, tag_t new_value) {
23         tag_t curr_value = CAS (old_value, new_value);
24         return (curr_value == old_value);
25     }
26
27     // returns true if serializer is scheduled, or false if not
28    bool produce (invocation_t* invocation);
29
30     // returns next invocation in serialization queue or NULL if empty
31    invocation_t* consume (void);
32 };
```

Figure 4.6: Serializer class

```
1  void serializer_t::produce (invocation_t* invocation) {
2      tag_t current_tag = tag;
3      bool scheduled = (tag != 0);
4
5      // Serializer was not scheduled before invocation produced
6      if (!scheduled) {
7          serialization_queue.produce (invocation);
8          set_tag (1);
9          return true;
10     }
11
12     // Serializer may have been scheduled before invocation produced
13     serialization_queue.produce (invocation);
14     scheduled = cas_tag (current_tag, current_tag + 1);
15
16     // Serializer no longer scheduled after invocation produced
17     if (!scheduled) {
18         // Invocation was consumed, no need to reschedule
19         if (queue.empty ()) return false;
20
21         // Invocation was not consumed, need to reschedule
22         else {
23             set_tag (1);
24             return true;
25         }
26     }
27
28     // Tag update succeeded, no need to reschedule
29     return false;
30 };
```

Figure 4.7: Serializer produce method

`false` if the invocation was successfully added to a scheduled task. The `produce` method begins by reading the current value of the tag. The subsequent actions are divided into two cases, depending on whether the tag is zero, indicating the serializer is not scheduled; or the tag is a positive integer, indicating the serializer is scheduled.

In the first case, the serializer is not currently scheduled (lines 6–10). The invocation is inserted into the serialization queue, and the tag of the serializer is set to one using the mutator method. Because the serializer is not scheduled, there can be no concurrent operations on the tag, so this unsynchronized update is safe. The `produce` method then returns `true`, indicating that a new extendable task should be instantiated and spawned for execution by the runtime system.

In the second case, the serializer is scheduled at the time the `produce` method begins, although it may be descheduled before the method completes. The method inserts the invocation object into the serialization queue (line 13), and then attempts to increment the tag using the `cas_tag` method (line 14). If the CAS update of the tag fails, it means that the serialization queue was drained and the tag was successfully set to zero by a concurrent `consume` operation. At this point, there are two possibilities, which are addressed by the code in lines 17–26. First, the invocation that was just inserted to the serialization queue may have been seen by the concurrent `consume`, in which case it would have been removed from the queue and executed. In this case, the queue will be empty (line 19) so `produce` returns `false`, because there is no need to reschedule the task. Second, if the queue is not empty, then the concurrent `consume` did not see the new invocation. The `produce` method handles this outcome on lines 22–25 by setting the tag to one and returning `true` to trigger creation and spawning of a new task.

Finally, if the tag update on line 14 is successful, then the method invocation was added to the serialization queue without a concurrent `consume` operation seeing an empty queue. The method invocation was therefore successfully appended to the current task, so the method returns `false` to indicate the task is still scheduled.

The `consume` method, listed in Figure 4.8, takes no parameters and returns a pointer to an invocation object, or `NULL` if the method finds the serialization queue empty and successfully deschedules the serializer. The first action of the `consume` method is to read the current tag of the serializer (line 2). Then, if it finds the queue empty, it attempts to deschedule the serializer (lines 4–7) by using the `cas_tag` method to reset the tag to zero (line 5). If the CAS is successful, the method returns `NULL` (line 6). If the CAS fails, then a concurrent `produce` succeeded in inserting a method invocation into the serialization queue. Whether

```
1  invocation_t* serializer_t::consume () {
2      tag_t current_tag = tag;
3
4      if (queue.empty ()) {
5          bool descheduled = cas_tag (current_tag, 0);
6          if (descheduled) return NULL;
7      }
8
9      invocation_t* invocation = queue.consume ();
10     return invocation;
11 }
```

Figure 4.8: Serializer consume method

the queue was initially non-empty, or the attempt to deschedule the serializer failed, when execution reaches line 9, there must be an invocation in the serialization queue. This invocation is removed from the queue and returned (line 10).

The tag used to synchronize concurrent `produce` and `consume` operations is implemented as a 64-bit integer, making overflow during the lifetime of a computer an unlikely event (Chase and Lev, 2005). However, since the tag is a finite value, it is still worthwhile to consider what will happen should the tag overflow. The only potential problem arises if a `consume` operation reads the tag and sees an empty serialization queue, and then a succession of `produce` operations cause the tag to wrap around to the same value seen by the `consume` operation. This would allow the `consume` operation to use CAS to set the tag to zero, indicating the serialization queue is empty, even though the queue still contains unexecuted method invocations. Aside from the improbability of the `consume` operation seeing the same value out of 2^{64} possibilities, it would require a multiple of 2^{64} method invocations to cause the tag to wrap around to the same value. The size of objects used to hold method invocations is at least 16 bytes on a 64-bit machine, and thus the system will run out of virtual address space long before 2^{64} method invocations could be inserted into the serialization queue. Therefore we do not expect tag overflow to ever occur during execution of a program.

The methods of the serializer class provide sufficient synchronization to ensure that `produce` and `consume` operations achieve consensus on the scheduling state of the serializer. In the next section, we use these primitives to implement the `delegate` and `quiesce` operations.

```
1  class extendable_task_t : public task_t {
2  private:
3      serializer_t* serializer;
4
5  public:
6      extendable_task_t (serializer_t* serializer) :
7          serializer (serializer)
8      {}
9
10     virtual void execute () {
11         invocation_t* invocation = serializer.consume ();
12         do {
13             bool executed = invocation->execute ();
14             if (!executed) break;
15             invocation->finish_multiple_delegation ();
16             delete invocation;
17             invocation = serializer.consume ();
18         } while (invocation != NULL);
19     }
20 };
```

Figure 4.9: An extendable task class for scheduling serializers

4.5 IMPLEMENTATION OF DELEGATE AND QUIESCE

Figure 4.9 lists the code for an extendable task class. Tasks are instantiated from this class and used to schedule serializers for execution. The `extendable_task_t` class has a single field, which stores a pointer to the serializer to be scheduled (line 3). The constructor initializes this field when the task is instantiated (lines 6–8). The serializer task implements the `execute` method (lines 10–19) to execute all the method invocations in the serialization queue. It does this by repeatedly calling the `consume` method on the serializer until it returns `NULL`. Let us first consider the case where the invocation returned by `consume` was only delegated to this serializer, i.e., it is not an aggregate operation. After removing an invocation object from the serializer, the task calls its `execute` method, which invokes the method using the receiver object and arguments passed to `delegate`, and then returns true to indicate the method was executed (line 13). The next two lines are irrelevant in the case of a method invocation delegated to a single serializer—the method invocation is always executed, so the task does not break out of the loop on line 13, and the call to `finish_multiple_delegation` on line 15 is a no-op.

```
1  template <typename C, typename B, typename... Args>
2  void delegate (C& obj, void (B::* method) (Args...), Args... args) {
3      // Get serializer associated with obj
4      serializer_t* serializer = obj.get_serializer ();
5
6      // Create an invocation object
7      invocation_t* invocation = new invocation_t (obj, method, args...);
8      // Add invocation to serializer
9      bool schedule = serializer.produce (invocation);
10
11     // If serializer is not currently scheduled, spawn a task
12     if (schedule) {
13         task_t* extendable_task = new extendable_task (serializer);
14         spawn (extendable_task);
15     }
16 }
```

Figure 4.10: Implementation of the `delegate` function

After the executing the invocation, the task deallocates it (line 16), and the loop continues.

Next, let us consider the case where the invocation returned from `consume` was the result of multiple delegation. This specialized invocation object contains a count of the number of serializers to which it was delegated. Each time that a serializer calls `execute` (line 13), this counter is atomically decremented. If the count is not zero after this decrement, it indicates that not all of the serializers involved in the invocation are ready, so `execute` returns false, causing the task to break out of the loop (line 14) and become descheduled. When the last serializer involved in the multiple delegation calls `execute`, the counter is decremented to zero, indicating that all serializers are ready to execute this invocation. This time, the method is invoked, and `execute` returns true. Then the last serializer calls `finish_multiple_delegation` (line 15), which reschedules any serializers with additional method invocations. The invocation is then deallocated (line 16) , and the serializer loop continues to the next invocation.

The extendable task is used to implement the `delegate` function, as shown in Figure 4.10. Its parameters of are the receiver object (`obj`), a pointer to the method to be invoked (`method`), and the arguments to the method (`args`). The `delegate` operation first identifies the serializer associated with the receiver (line 4). It then creates a new invocation object (line 7) to hold the receiver object, method

```
1  template <typename C>
2  void quiesce (C& object) {
3      // Get serializer associated with object
4      serializer_t* serializer = object.get_serializer ();
5
6      // Yield processor until serializer is empty
7      while (!serializer->get_tag () != 0) {
8          yield ();
9      }
10 }
```

Figure 4.11: Implementation of the `quiesce` function

pointer, and arguments. If any of the arguments is a private object, creation of this invocation also causes it to be delegated to the serializers associated with these objects to handle the case of multiple delegation. The `delegate` operation then inserts the resulting invocation object into the serialization queue using the `produce` method of the serializer (line 9). If `produce` returns false, the invocation extended an existing task. If `produce` returns true, it indicates that a new task must be scheduled to execute the serializer, so a new `extendable_task_t` object is instantiated and spawned (lines 12–15). Thus at the end of the `delegate` function, the invocation has been scheduled for execution, either by extending an existing task, or creating and spawning a new task.

The implementation of the `quiesce` function is listed in Figure 4.11. This function waits for all method invocations delegated to the serializer associated with an object to complete, so that dependent methods can safely be invoked on that object. It accepts the object to quiesce as its single parameter. The `quiesce` function first identifies the serializer associated with the object (line 4). It then repeatedly checks the tag in a loop (lines 7–9) until the tag is set to zero, indicating the task executing the serializer has completed. To avoid the deleterious effects of busy waiting (Mellor-Crummey and Scott, 1991), the loop calls the `yield` function inside the loop, which surrenders the processor so that other threads may run (Arora et al., 1998).

4.6 NONBLOCKING SERIALIZER SCHEDULING

In Section 4.1, we stated that a serializer should have a strong progress condition to ensure that a delay of one thread accessing the serializer does not hamper the

progress of another thread accessing the serializer. This goal is especially important in the context of a multiprogrammed operating system, because preemption of worker threads can introduce significant delays. Before analyzing the progress condition of the serializer, we first review the hierarchy of progress conditions for operations on a concurrent data structure.

The strongest progress condition is *wait-freedom*, which guarantees that every concurrent operation completes in a finite number of steps (Lamport, 1974; Herlihy, 1991). The next strongest progress condition is *lock-freedom*, which guarantees that at least one concurrent operation finishes in a finite number of steps (Herlihy, 1991). The power of these two properties is that they guarantee that a computation makes progress regardless of how threads are scheduled.

Herlihy and Shavit (2008, pg. 60) describe a weaker class of *dependent* progress conditions, which ensure progress only if the underlying system provides certain guarantees. *Isolation*—executing an operation to completion without any other operation taking steps—is an example of such a guarantee. A notable example of a dependent progress condition is *obstruction-freedom* Herlihy et al. (2003), which states than an operation will finish in a finite number of steps when executed in isolation.

The producer-consumer buffer used to implement the serialization queue (Figure 4.5) is wait-free—both the `produce` and `consume` operations will always complete in a finite number of steps (Hendler and Shavit, 2002). By inspection, we see that the `produce` and `consume` operations of the serializer are also wait-free, because they contain no loops, and no blocking operations. Furthermore, the CAS instructions in these operations will fail at most a single time. In the `produce` method (Figure 4.7), if the CAS on line 14 fails, a concurrent `consume` operation removed the last method invocation and descheduled the serializer task. In this case, the `produce` method indicates that the serializer should be rescheduled, completing in a finite number of steps. In the `consume` method (Figure 4.8), if the CAS on line 5 fails, then a concurrent `produce` has added another method invocation to the serializer. The `produce` method returns this invocation, completing in a finite number of steps. Thus neither the `produce` or `consume` operations require more than a single CAS instruction. This wait-free serializer implementation ensures that both delegation and serializer execution make progress when concurrently accessing a serializer.

While the serializer provides wait-free operations, the progress guarantees for the overall runtime depend on the other components in the system. The progress condition of `delegate` operation (Figure 4.10) is determined by the properties of the `new` operator (line 7) and the `spawn` operation (line 14). The progress condition

of the extendable task (Figure 4.9) is determined by the progress conditions of the runtime that schedules its execution and the `delete` operator. Thus the `delegate` operation and the serializer depend on the same two components to determine their progress conditions: the dynamic memory allocation system, and the runtime scheduler.

If a multiprocessor system provides a nonblocking dynamic memory allocation system, such as those proposed by Michael (2004) and Hudson et al. (2006), then we may use the default `new` and `delete` operators without modification to perform lock-free allocation and deallocation. As of this writing, nonblocking allocators are not widely available—most allocators still make heavy use of mutual exclusion locks. Therefore PROMETHEUS does not assume the presence of a nonblocking allocator, and instead overloads the C++ `new` and `delete` operators for `invocation_t` objects to use a lock-free recycling scheme similar to the one described by Hendler and Shavit (2002) for the PCB.

The final component contributing to the progress guarantees of serializer operations is the task scheduling system itself. While the particular guarantees provided by different schedulers vary, the lock-free scheduling algorithm described by Arora et al. (1998) and improved upon by Chase and Lev (2005) have been adopted by many runtimes, including the task scheduler targeted for Java 7 (Lea). The PROMETHEUS runtime described in Chapter 5 uses these algorithms to perform lock-free task scheduling.

In the past, many parallel programs were run on systems dedicated to their execution. Today, the ubiquity of multicore processors means that many future parallel programs will run in multiprogrammed environments. Composing our wait-free serializer implementation with lock-free memory allocation and task scheduling techniques yields nonblocking scheduling and execution of serializers, ensuring graceful performance degradation in the presence of heavily loaded systems.

4.7 SUMMARY

In this chapter, we identified the key elements of a serializer design: a serialization queue to track delegated method invocations and maintain their order, and a mechanism to schedule these method invocations for execution by worker threads. We also described a set of goals for a serializer design targeting general-purpose multiprocessors running a multiprogrammed operating system. We proposed a novel scheme for dynamic task extension, allowing serializers to exploit existing task scheduling support, while still ensuring the serialization of dependent

method invocations. For the remainder of the chapter, we detailed a nonblocking implementation of the PROMETHEUS serializer, and showed how the `delegate` and `quiesce` operations manipulate the serializer to coordinate a data-driven decomposition.

5 THE PROMETHEUS RUNTIME

> *In the case of non-determinism, I also believe this is an issue of abstraction. Under the hood the program is always going to be non-deterministic since there is no reasonable way to run a parallel program in lock-step in exactly the same way on every run. Beyond the technical problems, such lock-step execution would likely be terribly inefficient. The idea is therefore to abstract away the non-deterministic behavior so that the programmer only sees deterministic behavior and any non-determinacy is hidden in the run-time system.*
>
> — Guy Blelloch (2009)

In Chapter 4, we described a serializer implementation that uses dynamic task extension to ensure that the method invocations delegated to a particular serializer are assigned to a single task, and relies on runtime support to dynamically schedule different serializers for execution. This chapter describes the design and implementation of the PROMETHEUS runtime, which provides dynamic task scheduling support for C++ programs. Together, the task-based serializer and the PROMETHEUS runtime enables data-driven decomposition of C++ programs to exploit state-of-the-art dynamic scheduling techniques.

Scheduling is the process of assigning tasks to a set of *worker threads* for execution. As we explained in Section 4.2, a task is a lightweight mechanism for the asynchronous invocation of a single subroutine (i.e., a function or method), allowing the callee to continue executing without waiting for the subroutine to complete. Worker threads, or *workers*, are threads created by the runtime system that run a scheduling loop, which repeatedly attempts to acquire and execute a task. Dynamic scheduling balances the load of tasks assigned to each worker as the program runs, adapting to differences in the amount of work performed by each task, as well as variations in the speed of workers resulting from preemption by a multiprogrammed operating system. The PROMETHEUS runtime employs a work-stealing algorithm with provable time and space efficiency bounds (Blumofe and Leiserson, 1999), and nonblocking, dynamically sized data structures (Arora et al., 1998; Chase and Lev, 2005). We review work-stealing schedulers in Section 5.1.

The principal novelty of the PROMETHEUS runtime is that it supports unrestricted work-stealing in the form of a C++ library. Previous implementations of work-stealing that fully realize the efficiency guarantees of the scheduling algorithms, including Cilk (Frigo et al., 1998) and Cilk++ (Leiserson, 2009), rely on

compiler or other translation support. Previous library implementations of work stealing, including Intel's Threading Building Blocks (TBB) (Kukanov and Voss, 2007), compromise the efficiency guarantees of work stealing due to two limitations of the existing sequential languages they support. The first limitation, which we describe in Section 5.2, is the lack of support for storing the continuation of a subroutine invocation, which is required to achieve the space efficiency guarantee. The second limitation, which we explain in Section 5.3, is that sequential languages store activation records (or frames) on a stack. Most work-stealing libraries use this stack structure to hold the activation records for tasks, and consequently must restrict scheduling so that tasks using the same stack do not run concurrently. PROMETHEUS overcomes both of these limitations with system-specific assembly language routines that provide the necessary functionality not provided by C++.

In Section 5.4, we describe the data structures used to implement the PROMETHEUS runtime. We then describe the runtime operations that manipulate these data structures to realize dynamic scheduling in Section 5.5. Note that while the material in this chapter describes important aspects of the PROMETHEUS runtime library, it is largely orthogonal to data-driven decomposition. Readers wishing to bypass this discussion should consider reviewing the material on the DAG model of parallelism in Section 5.1, which provides background for Chapter 6.

5.1 TASK SCHEDULING ALGORITHMS

Task scheduling algorithms may broadly be divided into two categories—*work sharing* and *work stealing* (Herlihy and Shavit, 2008, pg. 381). Algorithms in both categories rely on a set of worker threads—normally one per hardware execution context—to execute tasks.[1] A worker typically maintains some form of work queue containing unexecuted tasks, from which it draws tasks to execute.

In a work-sharing algorithm, when a task is spawned, the scheduler decides whether the new task should be executed by the present worker, or whether the new task should be migrated to a different worker (Rudolph et al., 1991; Lüling and Monien, 1993; Hendler and Shavit, 2002). This decision involves assessing the worker's current load, usually measured by the number of tasks in its work queue, and comparing this with the load of other workers in the system. The main drawback of work sharing is that when all workers are heavily loaded, there is no benefit to performing load balancing. In this scenario, the time spent deciding whether to migrate each new task is wasted. Furthermore, the process of assessing the load

[1] Additional threads may be necessary when tasks perform considerable I/O.

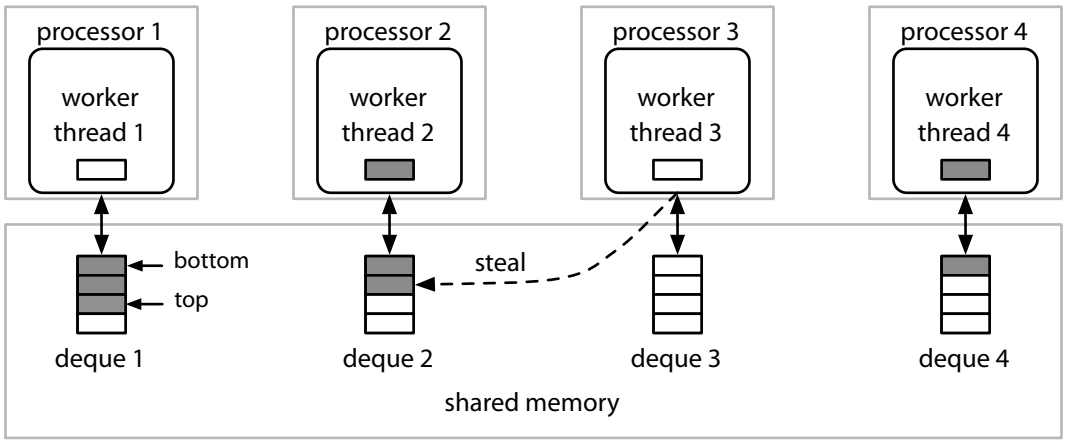


Figure 5.1: Scheduling tasks with work stealing

of other workers also incurs a significant amount of unnecessary communication.

Figure 5.1 depicts the operation of a work-stealing scheduler. Each processor executes a worker thread, and each worker is either currently executing a task (shown as a gray box inside the worker), or performing scheduling (shown as a white box inside the worker). Each worker maintains a double-ended queue, or *deque* of tasks. The deque has two ends: the local worker pushes and pops tasks on the *bottom* end, and other workers may steal tasks from the *top* end. (We use the standard definitions of bottom and top for the ends of the deque, which results in results in the deqeus in Figure 5.1 appearing to be upside-down—the bottom is the end closes to the worker threads.) Because the work-stealing deque only allows insertions at the bottom end, it is not a general deque, but an *input-restricted* deque (Knuth, 1997). A worker pushes new tasks into the bottom of its deque, and when it completes a task, it performs a scheduling operation to get the next task. This scheduling operation first attempts to pop the next task from the worker's local deque. If this deque contains tasks, like the deque for worker thread 1 in Figure 5.1, then the worker pops this task from the deque and executes it.

If the local deque is empty, like the deque of worker thread 2 in Figure 5.1, then the worker becomes a *thief* and the scheduling operation attempts to steal a task from another worker. It randomly selects another worker to be a *victim*, and then accesses the top of the victim's deque to steal a task. (Again, in Figure 5.1, the top is the end furthest from the worker thread.) In Figure 5.1, worker thread 2 selects worker thread 1 as the victim, and since the deque of worker thread 1 is not

empty, worker thread 2 will successfully steal a task. If the victim's deque is empty, the scheduling operation will wait for some amount of time and then attempt to steal from another randomly selected victim.

Work-stealing algorithms result in less communication and synchronization than work-sharing algorithms, because busy worker spend no time performing scheduling operations. Instead, the brunt of scheduling overhead is borne by idle workers. These advantages were observed empirically in early implementations of work stealing. Burton and Sleep (1981) developed an early incarnation of work stealing to schedule parallel execution of programs written in functional languages. Halstead (1985) used work stealing to schedule parallel work generated by the future construct of Multilisp. Later work on the Multilisp project by Mohr et al. (1991) introduced the use of a deque as the work queue. Work-stealing schedulers were popularized by Cilk (Blumofe et al., 1995; Frigo et al., 1998), a task programming extension to the C language. While optimal scheduling is NP-complete (Garey and Johnson, 1979), the Cilk project proved tight bounds on both the time and space required for work-stealing schedulers (Blumofe and Leiserson, 1999).

The DAG Model of Parallel Computation

To facilitate the presentation of the efficiency of work-stealing schedulers, we first review the directed acyclic graph (DAG) model developed by Eager et al. (1989) to quantify the parallelism of a dynamic computation. We describe this model using the more contemporary terminology of Cormen et al. (2009, chap. 27). The DAG model represents program execution as a graph, with vertices representing dynamic instructions, and edges between the vertices representing dependences between these instructions. Figure 5.2 depicts an example DAG. The number in each node is given for the purpose of identification, and does not have any other significance.

There are two important metrics used to quantify the performance of the program using the DAG model. The first metric is *work*—the minimum execution time of the program running on a single idealized processor that executes one instruction per cycle. The number of nodes in the DAG of a parallel computation determines its work, denoted T_1. For the example in Figure 5.2, $T_1 = 14$.

If we execute a program in parallel on an computer with N idealized processors, the program could complete at most N instructions per cycle. This gives a lower

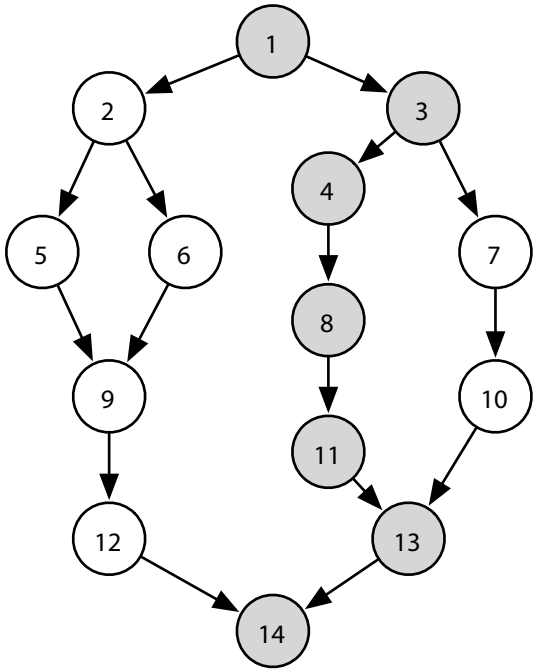


Figure 5.2: An example DAG representing parallel execution

bound on the execution time T_N of the parallel program called the *work law*:

$$T_N \geqslant T_1/N \,. \tag{5.1}$$

The second metric is *span*, which is the execution time of the program running on an infinite number of idealized processors. The longest path through the DAG, or *critical path*, determines the span of the computation, denoted T_∞. The critical path of the computation represented by Figure 5.2 is highlighted in gray, and counting the length of this path gives us $T_\infty = 7$. Because the critical path cannot be parallelized, it gives us another lower bound on the execution time of this program on N processors, called the *span law*:

$$T_N \geqslant T_\infty \,. \tag{5.2}$$

Using work and span, we may define the *average parallelism* P of the program's

execution as the speedup when run on an infinite number of processors:

$$P = T_1/T_\infty \,. \tag{5.3}$$

The example in Figure 5.2 has work $T_1 = 14$ and span $T_\infty = 7$, and thus has parallelism $P = 14/7 = 2$. The implication of parallelism is that a parallel program can only achieve perfect linear speedup for a number of processors $N \leqslant P$. In other words, the marginal benefit of adding additional processors will decrease once the number of processors N is larger than the parallelism P.

The Efficiency of Work Stealing

The work and span laws provide a lower bound on the execution time of a parallel program. Using the DAG model, Blumofe and Leiserson (1999) show that the Cilk work-stealing algorithm gives an upper bound:

$$T_N \leqslant T_1/N + O(T_\infty) \,. \tag{5.4}$$

Equation 5.4 says that the execution time of the program on P processors will be less than the work divided by the number of processors N (the work term) plus a critical path term $O(T_\infty)$.

Recall that the parallelism of the program is the speedup on an infinite number of processors. Now let the *parallel slackness* be the ratio of parallelism to the number of processors: P/N. Frigo et al. (1998) show that if the program has ample parallel slackness, i.e., the average parallelism of the program is significantly greater than the number of processors:

$$P = T_1/T_\infty \gg N. \tag{5.5}$$

Rearranging this equation, we have:

$$T_\infty \ll T_1/N. \tag{5.6}$$

Thus the assumption of parallel slackness makes the critical-path term of Equation (5.4) negligible in comparison with the work term. This gives us the following upper bound on the execution time:

$$T_N \leqslant T_1/N. \tag{5.7}$$

Taking equation (5.7) in combination with the work law, Equation (5.1) gives us:

$$T_N \approx T_1/N \,. \tag{5.8}$$

Thus under the assumption of parallel slackness, a work-stealing scheduler gives perfect linear speedup. This result provides a firm theoretical basis for assigning tasks to worker threads. The provable efficiency of work-stealing has resulted in its widespread adoption: In addition to Cilk and Cilk++ (Leiserson, 2009), work-stealing algorithms are used today in libraries such as Intel Threading Building Blocks (TBB) (Kukanov and Voss, 2007), the Java Fork/Join framework (Lea, 2000), Microsoft's .NET Task Parallel Library (TPL) (Leijen et al., 2009), and recent languages including Chapel (Chamberlain et al., 2007), Fortress (Allen et al., 2007), and X10 (Charles et al., 2005).

Drawbacks of Work Stealing

While work-stealing provides provably efficient scheduling under the assumption of parallel slackness, it does have notable shortcomings. The first drawback is that work-stealing schedulers do not account for affinity that tasks may have for particular worker threads or processors. Frigo et al. (1999) show that divide-and-conquer algorithms naturally exploit cache locality without intervention from the programmer, but many applications require other kinds of algorithms. If a set of tasks accesses spatially local data, it may be beneficial to map these tasks to the same worker thread so that they benefit from cache locality. Applications may also benefit from task affinity when running on cache-coherent non-uniform memory access (CCNUMA) servers (Conway and Hughes, 2007; Intel, 2009b). Assigning tasks to processors closest to the memory they access can improve performance by reducing the average memory access time. Unfortunately, most current task programming systems do not support affinity in their API, and randomized work-stealing schedulers are inherently oblivious to locality.

Recent research suggests it may be possible to remedy the lack of affinity support in work-stealing schedulers. Acar et al. (2000) describe a mechanism for locality-guided work stealing, but their proposal greatly increases the number of expensive atomic operations required for each scheduling operation. Contreras and Martonosi (2008) suggest initially assigning tasks to specific processors, and then falling back on work stealing for load balancing, but they do not evaluate the performance or prove the efficiency of this approach.

The second drawback with work stealing is that its time-efficiency guarantee breaks down in the absence of parallel slackness. Contreras and Martonosi (2008) and Guo et al. (2009) both empirically observe that for some applications, work stealing scales poorly to large numbers of worker threads. Contreras and Martonosi attribute this to the fact that the increasing number of workers reduces the probability that a random steal will select a victim with a large amount of work. They suggest occupancy-based stealing, which selects a victim with the most work as a remedy for this issue. Guo et al. observe that work stealing performs poorly for long chains of sequential spawns when using a large number of workers. In both these examples, the underlying limitation is that the parallel slackness is insufficient to maintain linear scaling.

Despite these shortcomings, we believe that a work-stealing scheduler is the best choice for the PROMETHEUS runtime because it has been thoroughly studied and rests on a firm theoretical basis. We note that data-driven decomposition using the task-based serializer, described in Chapter 4, may partially remedy locality issues, because it allows programmers to establish pipelines of operations applied to a private object, which are likely to execute on the same processor. We plan to continue evaluating the interaction of data-driven decomposition with scheduling algorithms as the number of hardware contexts in multi-core processors continues to grow.

5.2 TASK CREATION

There are two possible ways to handle the creation of new tasks in a work-stealing scheduler: *eager task creation*, and *lazy task creation*. With eager task creation, a worker spawns a task by pushing it into the bottom of the work deque and then continues executing the code after the spawn (i.e., the continuation of the task). When the worker reaches a `sync` directive, it begins drawing tasks from the bottom of its local deque and executing them. In the meantime, other workers may steal tasks from the top of the deque and execute them.

Eager task creation is straightforward to implement, but it can result in an explosion in memory consumption when a program spawns a large number of tasks. Consider the code in Figure 5.3, which lists a simplified version of the `blackscholes` benchmark from the PARSEC suite (Bienia et al., 2008). This program uses the Black-Scholes formula to compute prices for a portfolio of European options. The program reads option data from a file into an array (line 2), and then loops over each array element (lines 3–6). It creates a task to compute the price of each option in the array (line 4) and spawns the task (line 5). After the loop, the

```
1 vector <option_t> options;
2 initialize_option_data (&options);
3 for (int i = 0; i < 100000000; ++i) {
4     task_t* task = new option_task_t (options[i]);
5     spawn (task);
6 }
7 sync ();
```

Figure 5.3: Simplified code for `blackscholes` benchmark

program performs a `sync` to wait for the tasks to complete (line 7).

A sequential version of this program (eliding the `spawn` and `sync` directives) would only require memory for one task at a time. Using eager task creation, the parallel version would require memory for as many tasks as there are options in the input, and would require the deques be large enough to store all of the tasks. Running the `blackscholes` example with its input of one hundred million options would result in a flood of tasks that may overwhelm the system.

Recognizing the problems with eager task creation, Mohr et al. (1991) developed lazy task creation for scheduling Multilisp programs. When a worker spawns a task using lazy task creation, it saves the continuation of the task, rather than the task, in its deque. The worker then executes the task itself. Once the task completes, it attempts to pop the continuation from the deque to resume execution.

While a worker is executing a task, it is possible that another worker may steal the task's continuation. The stealing worker will execute the continuation until it encounters another spawn point, at which point it will save a new continuation into its deque and commence execution of the new task. This process continues, with each worker peeling off a single task for execution, until a worker reaches the `sync` directive. Applying this approach to our example limits the number of active tasks to the number of active workers, so executing `blackscholes` on a machine with N processors would require storage for N tasks, rather than one hundred million, and would require only one entry in each deque to hold the continuation.

Blumofe and Leiserson (1999) prove that lazy task creation bounds the memory consumption of a program that consumes space S_1 in a single-processor execution consumes at most $S_1 N$ space in an N-processor execution. This finding motivated the adoption of lazy task creation in both the Cilk (Frigo et al., 1998), and Cilk++ (Leiserson, 2009) languages. However, because most popular sequential languages do not provide a mechanism for saving a continuation, many recent task scheduling libraries, including Intel TBB (Kukanov and Voss, 2007), the Java Fork/Join

framework (Lea, 2000), and the Microsoft .NET TPL (Leijen et al., 2009) use eager task creation.

Task scheduling frameworks using eager task creation alleviate the memory explosion problem by providing special looping constructs such as `parallel_for` and `parallel_while`, which recursively subdivide loops in a divide-and-conquer fashion.[2] These constructs reduce memory consumption so that it is logarithmically proportional, rather than linearly proportional, to the number of tasks. However, they are only applicable to looping code, and thus pathological programs with excessive memory consumption are still possible when using eager task creation.

Data-driven decomposition strives to use existing sequential programming constructs and semantics. Requiring programmers to reason about the memory consumption of parallel operations and potentially requiring them to restructure code to reduce its memory footprint would severely undermine these goals. Therefore the PROMETHEUS runtime implements lazy task creation, ensuring an upper bound on memory consumption. (To the best of our knowledge, PROMETHEUS is the first library implementation of a work-stealing scheduler to use lazy task creation.) Because C++ does not support saving the continuation of a method invocation, PROMETHEUS includes a set of functions for saving and restoring continuations implemented in assembly language, which we detail in Sections 5.4 and 5.5.

5.3 MANAGING TASK ACTIVATION RECORDS

One of the foremost challenges to implementing support for tasks in an existing sequential language is managing the activation records of the asynchronous subroutine (function or method) invocations performed by tasks. An *activation record* (sometimes called a *frame*) stores the data used to invoke a subroutine, including the return address and the values of arguments and local variables. Sequential languages use a linear *call stack*, which allocates activation records in a contiguous region of memory. When a function is invoked, it pushes an activation record onto the stack, and when it returns, it pops the activation record off of the stack. These operations are extremely efficient, requiring only a simple addition or subtraction to the stack pointer, which is usually stored in a dedicated register. The fact that a routine in a sequential language directly invokes at most one subroutine at any

[2] These constructs are also useful in systems that employ lazy task creation, such as Cilk++, because they increase the parallel slackness, reducing the critical path contribution to the execution time.

```
1  void A () {
2      spawn (&B);
3      C ();
4      sync ();
5  }
6
7  void B () {
8      spawn (&D);
9      E ();
10     sync ();
11     foo ();
12 }
13
14 void C () {
15     spawn (&F);
16     G ();
17     sync ();
18 }
```

Figure 5.4: Example of parallel function calls in a task-based program

given time enables this simple scheme.

Tasks execute subroutines asynchronously, allowing a particular routine to directly invoke multiple subroutines simultaneously. Figure 5.4 lists an example of parallel function calls in a task-based program, due to Frigo (2009). This example code corresponds to a simple traversal of a balanced binary tree with four leaves. The function `A` spawns function `B` (line 2) and then invokes function `C` (line 3) for (potentially) parallel execution. (Note that there is no benefit to spawning `C`, because it is immediately followed by a `sync`.) A linear stack is inadequate for this code, because at most one of the activation records of `B` and `C` can be placed on the stack after the activation record for `A`.

Previous work-stealing schedulers have adopted one of two solutions to this problem (Taura et al., 1999). The first solution is to use compiler or other translation support to implement a more appropriate mechanism for allocating activation records. Examples of this approach include Cilk (Frigo et al., 1998), Cilk++ (Leiserson, 2009), Lazy Threads (Goldstein et al., 1996), and StackThreads/MP (Taura et al., 1999). While this completely solves the problem of managing activation records, the resulting code cannot interoperate with existing code that uses stack-allocated activation records.

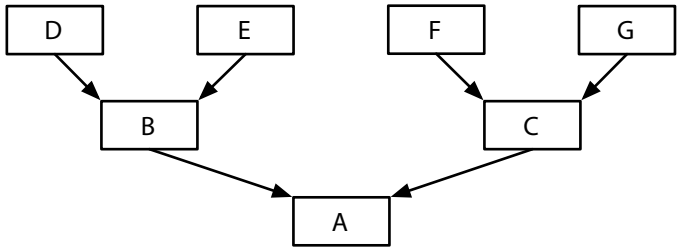


Figure 5.5: Cactus stack for the code in Figure 5.4

The second solution is to allocate the activation record of a particular task on the linear stack of the worker executing that task. All current task scheduling libraries, including TBB (Kukanov and Voss, 2007), the Java Fork/Join framework (Lea, 2000), and the Microsoft .NET TPL (Leijen et al., 2009) use this approach to maintain interoperability with existing code. As we will see later in this section, allocating task activation records on linear stacks can lead to several pathological behaviors. Kukanov and Voss (2007) describe the mechanisms used to alleviate these problems in TBB, which either require extra effort on the part of the programmer; or restrict scheduling, so that workers may sometimes go idle while there are still unexecuted tasks.

We will now examine these two approaches in detail. We use Cilk++ and TBB for our examples, since they provide detailed documentation of their handling of activation records. At the end of the section, we describe the PROMETHEUS approach to managing activation records, which provides the full compatibility with existing code of the library approaches, while not requiring any restriction on scheduling or extra programmer effort.

The Cilk++ Approach: Cactus Stack

Cilk++ replaces the linear call stack with *cactus stack* (also known as a *saguaro stack* or *spaghetti stack*) (Hauck and Dent, 1968). Each function invocation, whether directly invoked or spawned as a task, allocates an activation record on the heap. Each activation record contains a pointer to the activation record of the calling function, so that the resulting structure resembles a tree or saguaro cactus. Figure 5.5 shows the cactus stack for the code in Figure 5.4 at the point when the functions `B`–`G` have been spawned or invoked, but have not yet completed.

Using a cactus stack allows Cilk++ to avoid the problem of mapping activation

records onto a linear call stack. However, replacing the linear stack with a different structure requires changes to the *calling convention* (sometimes called the *calling sequence*)—the series of events that sets up an activation record when a subroutine is invoked. Calling conventions are determined by the *application binary interface* (ABI) of a particular system, which specifies how applications should interact with libraries, other applications, and the operating system. Therefore, changing the calling convention has the following consequences: First, it requires a compiler modified to generate the necessary function prologue and epilogues to create and destroy activation records, respectively. Second, Cilk++ functions are distinct from other C++ functions, and these two kinds of functions cannot call each other because they handle activation records differently. This makes it difficult to integrate Cilk++ code into an existing program—either the entire code base must be converted to Cilk++, which is impossible if third-party libraries are used; or the program must be split into Cilk++ and C++ components, with extra code added to convert between calling conventions at the boundaries of these components (Cilk Arts, 2008).

The TBB Approach: Linear Stacks

The purpose of software libraries is to provide functionality that can be reused in different programs, so they must use standard calling conventions. Therefore, previous library implementations of work-stealing schedulers, including TBB (Kukanov and Voss, 2007), use the linear stack structure of the sequential languages they support. This approach can result in serious problems: Frigo (2009) describes how a naive implementation of work-stealing using linear stacks can degrade the operation of the scheduler, and potentially lead to stack overflow.

We restate this example of Frigo below, assuming a work-stealing scheduler with lazy task creation, operating with three worker threads. We follow the sequence of operations of the code from Figure 5.4. Initially, worker 1 is executing function `A`. One possible execution of the remainder of the program is as follows:

1. `A` spawns `B` (line 2). Worker 1 saves the continuation of `B` in its deque and invokes `B`.
2. Worker 2 steals the continuation of `B`.
3. The continuation `B` (in function `A`) invokes `C` (line 3) in worker 2.
4. `C` spawns `F` (line 15). Worker 2 pushes the continuation of `F` in its deque and invokes `F`.
5. Meanwhile, `B` spawns `D` (line 8). Worker 1 saves the the continuation of `D` in its deque and invokes `D`.

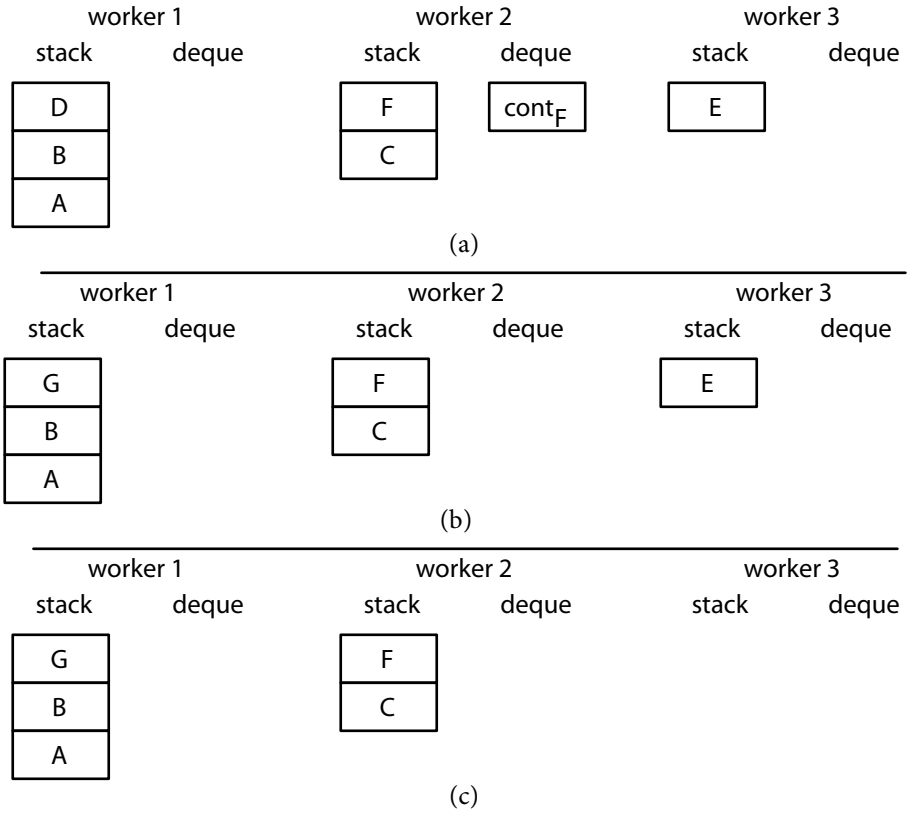


Figure 5.6: A possible execution of the code in Figure 5.4 with a linear stack

6. Worker 3 steals the continuation of `D`.
7. The continuation of `D` (in function `B`) invokes `E` (line 9) in worker 3.

This point in the execution is depicted in Figure 5.6(a). Worker 1 has `A`, `B`, and `D` on its stack. Worker 2 has `C` and `F` on its stack and the continuation of `F` in its deque. Worker 3 has `E` on its stack. Execution continues as follows:

8. Worker 1 completes execution of `D` and pops its activation record off of the stack. Finding its deque empty, it steals the continuation of `F` from worker 2.
9. The continuation of `F` (in function `C`) invokes `G` (line 16) in worker 1.

Figure 5.6(b) shows this stage of execution. Now we see a problem beginning to emerge—the activation record for `G` is sitting on top of the activation record of `B`, which is not its ancestor in the dynamic call graph. The consequences of this

```
1  void B1 () {
2      spawn (&D);
3      spawn (&E);
4      set_continuation (&B2);
5  }
6
7  void B2 () {
8      foo ();
9  }
```

Figure 5.7: Example of TBB continuation tasks (syntax simplified)

problem manifest in the next step of the example. Worker 3 completes execution of E and pops its activation record off of the stack, returning to line 10 of B. Since D has completed, we expect B to continue executing past the `sync` to the invocation of `foo` on line 11. However, this is not safe, because the activation record of B is on the stack of worker 1, underneath the activation record of G. If worker 3 did continue executing B from the continuation of E, the invocation of `foo` would overwrite the activation record of G and cause the program to fail. Therefore the scheduler cannot execute `foo` until G completes, including all functions that it calls or spawns.

This example shows how mapping the tree-like structure of task activation records onto a linear stack can trap an activation record beneath an activation record of a function that is not its descendent, preventing it from continuing execution. This is a significant problem, since it can result in idle workers when there is still work to be done, but an even worse outcome is possible—if this scenario occurs repeatedly on a worker's stack, it may overflow and crash the program.

Kukanov and Voss (2007) describe two techniques that Intel TBB uses to mitigate these problems. First, the scheduler only allows a worker to steal a task that is more deeply nested than the tasks currently on its stack. This policy reduces the probability of overflowing the stack, because more deeply nested tasks are likely to have fewer child tasks, and thus require less stack space. Second, the programmer can divide the original task into a task that returns before its children complete, so that its stack space can be reclaimed; and a *continuation task*, which executes once all the child tasks finish. For example, the programmer might divide the task for function B into a task B1 that spawns D and E, and a continuation task B2 that calls `foo`, as shown in Figure 5.7. Here the programmer specifies that the continuation

task B2 should execute on completion of D and E (line 4). The spawn of B on line 2 of Figure 5.4 is then replaced with a spawn of B1. When B1 executes, it spawns D and E, and returns without waiting for these children to complete, popping its activation record off of the stack. Then when tasks D and E complete, the runtime schedules B2 for execution.

While the TBB techniques partially alleviate the problems of storing task activation records on linear stacks, they introduce two disadvantages. First, TBB may not always be able to exploit parallelism in the program due to the scheduling restrictions. Second, continuation tasks require significant extra programmer effort to identify tasks to split, and to manually divide the function or method into a task and one or more continuation tasks. This process requires the programmer to understand how TBB uses the stack and structure their program accordingly, lowering the level of abstraction provided by task programming.

The Prometheus Approach: Recycled Stacklets

Like TBB, PROMETHEUS is also implemented as a library, but takes a completely different approach to managing activation records, inspired by the strategy used in the Hood user-level threading library (Papadopoulos, 1998). This strategy provides each running task with a *stacklet*—a linear stack that stores the activation records of ordinary subroutine invocations for that particular task. We will show that stacklets allow PROMETHEUS to perform unrestricted scheduling, while maintaining the compatibility benefits of using the standard calling convention.

We list the actions of the PROMETHEUS runtime performs for all events that allocate or deallocate activation records below:

Subroutine invocation. When invoked, a subroutine allocates an activation record on the stacklet using the standard calling convention specified by the system ABI. This typically involves incrementing the stack pointer by the size of the activation record.

Subroutine return. When returning, a subroutine deallocates the current activation record on the stacklet using the standard calling convention specified by the system ABI. This typically involves restoring the previous value of the stack pointer, which is either stored in the activation record or a designated register.

Task spawn. To spawn a task, a worker saves the continuation into its local deque, including a pointer to the current stacklet. The worker then allocates a new stacklet, and changes its stack pointer to point to the new stacklet and then

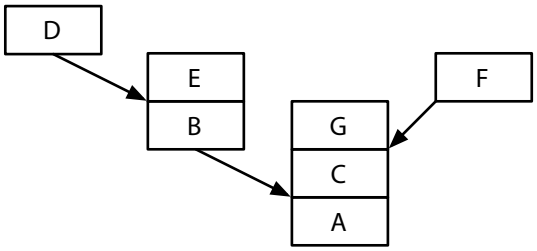


Figure 5.8: Stacklets for the code in Figure 5.4

invokes the subroutine (function or method) for the task.

Task completion. When a task completes, the worker deallocates its stacklet. If there is a continuation in the local deque, the worker retrieves it, switches to its stacklet, and resumes its execution. If the local deque is empty, the worker restores its original call stack, and attempts to steal work from another worker.

Continuation steal. When a worker successfully steals work, it saves its current stack pointer into a reserved location, and then sets its stack pointer to the stack pointer of the continuation. The worker then resumes executing the continuation.

Figure 5.8 shows the stacklets for the code in Figure 5.4, at the point where `A` through `G` have been invoked, but none have completed. Note that each spawned function (`B`, `D`, and `F`) begins a new stacklet, while the original stacklet remains with the continuation of the spawn. Ordinary function invocations (`C`, `E`, and `G`) push an activation record on the existing stacklet.

Stacklets form a cactus stack that is more coarse-grained than the Cilk-style cactus stack. Cilk allocates every activation record on the heap for both subroutines spawned as tasks and ordinary subroutine invocations. Even using an optimized memory allocator (Frigo et al., 1998), heap-based activation record allocation still incurs more overhead on each subroutine invocation than the linear stack mechanism. By contrast, PROMETHEUS uses the standard stack-based calling convention to allocate activation records in the current stacklet for ordinary subroutine invocations, so they do not incur any extra overhead.

PROMETHEUS allocates new stacklets using `mmap`, a relatively expensive system call. To prevent this overhead from becoming a bottleneck, we borrow another technique from Hood: each worker maintains a local pool of stacklets. When a worker requires a stacklet to execute a task, it first tries to reuse one from the pool, and only allocates a new stacklet if the pool is empty. When the task is complete,

the stacklet is recycled back into the worker's pool. Once a worker reaches the maximum depth of the task graph, it never needs to allocate another stacklet. Using this combination of fast linear-stack-based allocation of activation records for ordinary subroutine calls with a scheme that amortizes the overhead of stacklet allocation over many tasks, PROMETHEUS is able to efficiently support the tree structure of task activation records.

In the PROMETHEUS runtime, an activation record is always a direct descendent of the previous activation record in a stacklet. The phenomenon of activation records becoming trapped underneath unrelated activation records that occurs when using a linear stack, as TBB does, never happens with stacklets. Consequently, PROMETHEUS can schedule tasks without any restriction of the work-stealing algorithm. When a worker steals a continuation, it also acquires the stacklet associated with the continuation, and adjusts its stack pointer to point to this stacklet before resuming the continuation. Avoiding the linear stack problem also eliminates the need for programmers to specify continuation tasks. This is especially important for PROMETHEUS, because exposing this artifact of control-driven decomposition in the API for data-driven decomposition would be highly undesirable.

Goldstein et al. (1996) originally proposed stacklets for the Lazy Threads execution model. Lazy Threads use a specialized calling convention, relying on a modified version of the GCC compiler to generate native code that adheres to these conventions. By contrast, PROMETHEUS uses the standard calling convention specified by the ABI of the host system. When a task is spawned, PROMETHEUS allocates a new stacklet, then uses a system-specific routine to change the stack pointer to point to the new stacklet before invoking the method. This mechanism, which we will describe in detail in the next two sections, allows PROMETHEUS to exploit the benefits of stacklets without any special compiler or pre-processor support, and maintains compatibility with existing software and libraries.

5.4 RUNTIME STRUCTURES

As we explained in the previous sections, the PROMETHEUS runtime is a C++ library that implements dynamic task scheduling via work-stealing with lazy task creation. Currently the library targets 32- and 64-bit x86 and SPARC architectures. To facilitate portability, system-specific code is isolated in a few architecture-specific files and a handful of assembly-language routines. In this section, we describe the data structures used to implement the runtime, and in the next section we describe how the runtime uses these structures to schedule tasks.

Threads

The PROMETHEUS runtime instantiates objects of the `thread_t` to serve as workers for scheduling. This class abstracts away all system-specific details of thread management, and provides an interface for common thread operations such as creation, destruction, and joining. The thread class also encapsulates management of thread-local storage (TLS).

PROMETHEUS currently supports Linux for x86 architectures and Solaris for SPARC architectures. For both of these platforms, the `thread_t` class uses POSIX threads (pthreads) for its implementation. Because the thread class encapsulates all threading functionality, porting PROMETHEUS to other threading implementations—such as Solaris threads or Windows API threads—would involve a straightforward replacement of pthread calls with the corresponding calls in the new API.

Stacklets

Figure 5.9 lists the code for the `stacklet_t` class. This class comprises two fields, one that indicates the size of the stacklet (line 3), and another that points to the location of the allocated stacklet (line 4). The constructor (lines 7–12) takes a single parameter specifying the size of the stacklet. It uses the `mmap` system call to allocate a chunk of memory of the desired size. The arguments to `mmap` specify that the memory pages should be mapped with read and write permissions; that they should be private to the current process; and that they should be anonymous, i.e., not associated with any file. Likewise, the destructor (lines 14–16) uses the `munmap` system call to deallocate the pages.

The `get_sp` method (18–21) is used to obtain a pointer into the allocated chunk of memory suitable to be used as a stack pointer. Stacks grow down (from higher addresses to lower addresses) on all platforms currently targeted by PROMETHEUS, so this pointer is obtained by starting with the top of the allocated block. However, these systems all have their own application binary interface (ABI) that specifies a required alignment for the stack on entry to a function call. Therefore `get_sp` calls a system-specific `get_abi_alignment` function to determine an offset from the top of the allocated memory chunk that will yield the correct alignment (line 19). This offset is then used to compute the highest correctly aligned address in the allocated storage, and this is returned as the value to use as a stack pointer (line 20).

```
1  class stacklet_t {
2  private:
3      size_t size;
4      void* storage;
5
6  public:
7      stacklet_t (size_t size) :
8          size (size)
9      {
10         storage = mmap (NULL, size, PROT_READ | PROT_WRITE,
11                         MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
12     }
13
14     ~stacklet_t () {
15         munmap (storage, size);
16     }
17
18     void* get_sp (void) {
19         size_t offset = get_abi_alignment ();
20         return ((char*) storage + size - offset);
21     }
22 };
```

Figure 5.9: Stacklet class

Termination-Detecting Barriers

The sync operation in task programming effects a local barrier that causes the caller to wait until all outstanding tasks complete. A *termination-detecting barrier* tracks outstanding tasks and recognizes their completion (Flood et al., 2001). (For the sake of brevity, we will refer to a termination-detecting barrier as simply a barrier, but it should not be confused with barriers used for global thread synchronization, as described by Mellor-Crummey and Scott (1991).) Conceptually, this barrier simply maintains a *join counter* that is incremented before a task is spawned and decremented when a task completes. When the join counter reaches zero, it indicates there are no outstanding tasks.

Figure 5.10 gives the PROMETHEUS implementation of a termination-detecting barrier. Rather than using a single counter, which would require synchronization of concurrent accesses and could potentially lead to cache contention, each barrier maintains an array of per-thread counters (line 3). We do not show this in the

```
1  class td_barrier_t {
2  private:
3      int64_t* counters;
4
5  public:
6      barrier_t () :
7          counters (new int64_t[prometheus::get_num_threads ()])
8      {}
9
10     ~barrier_t () {
11         delete[] counters;
12     }
13
14     void add_tasks (int64_t num_tasks) {
15         prometheus_tid_t tid = get_thread_id ();
16         counters[tid] += num_tasks;
17     }
18
19     void subtract_tasks (int64_t num_tasks) {
20         prometheus_tid_t tid = get_thread_id ();
21         counters[tid] -= num_tasks;
22     }
23
24     bool complete () {
25         int64_t num_tasks = 0;
26         for (int i = 0; i < prometheus::get_num_threads (); ++i) {
27             num_tasks += counters[i];
28         }
29
30         return (num_tasks == 0);
31     }
32 };
```

Figure 5.10: Termination-detecting barrier

simplified code of Figure 5.10, but these counters are each aligned on a cache-line boundary to avoid false sharing. The constructor (lines 6–8) allocates memory for this array and initializes the counters to zero. Correspondingly, the destructor (lines 10–12) deallocates this memory.

The barrier provides a method to add to the task count when tasks are spawned (lines 14–17), and another method to subtract tasks when they complete (lines 19–22). These methods take the count of tasks to add or subtract as their lone parameter, and after determining the identity current thread, increment or decrement the counter for that thread. Because each counter in the array is written by a single thread, these methods require no synchronization.

The thread-local counters are represented as signed integers because they may be positive or negative, depending on how many tasks a particular thread has spawned and completed. The number of outstanding tasks is determined by sum of all the counters. The `complete` method (lines 24–31) detects termination by looping over the array (lines 26–28) and adding together the value of each counter. The method returns true if this value is equal to zero, and false otherwise.

Note that `complete` uses no synchronization, and so the values of a particular local counter may change while the sum is being computed, either before or after it is read by the loop. As we shall see in the next section, the runtime increments the task count before the task is spawned, which ensures that the total count is always positive while there are still outstanding tasks. (If the counter were incremented after the task was spawned, it is possible that the thread executing the task could complete its execution and decrement its counter before the spawning thread performed the increment, causing the total counter value to be less than it should be.) Therefore it is impossible for `complete` to return true while there are still outstanding tasks. It is possible for `complete` to return false when all tasks have completed, but this cannot cause erroneous execution, and termination will be detected on the next call.

Contexts

PROMETHEUS maintains information about the state of running and suspended computations using *contexts*. Contexts serve multiple purposes: they provide the ability to save and restore continuations, as well as enabling transitioning between the scheduler and program code. Figure 5.11 lists the `context_t` data structure, which comprises three fields. The first field is a pointer to a termination-detecting barrier (line 2), which is used to track the number of outstanding tasks for an executing context. The remaining fields are used to save the state of a suspended

```
1  struct context_t {
2      td_barrier_t* td_barrier;
3      stacklet_t* stacklet;
4      void* save_sp;
5
6      context_t () :
7          save_sp (NULL), stacklet (NULL),
8          td_barrier (new td_barrier_t)
9      {}
10
11     ~context_t () {
12         delete td_barrier;
13     }
14 };
```

Figure 5.11: Context structure

```
1 void* save_and_move_context (void** save_sp, void* new_sp);
2 void* restore_context (void* new_sp, void* ret_val);
3 void* save_and_restore_context (void** save_sp, void* new_sp,
4                                 void* ret_val);
```

Figure 5.12: Context management functions

context. The second field is a pointer to the stacklet object for this context (line 3). The constructor (lines 6–9) and destructor (lines 11–13) allocate and deallocate this barrier, respectively. The third field (line 4) is used to store the current stack pointer when a context is suspended.

The standard C library provides the `getcontext` and `setcontext` routines to manage user-level contexts. However, using these routines would require two system calls on every context swap, making these routines significantly slower than a purely user-space solution. To enable PROMETHEUS to profitably parallelize more finely grained tasks, we have implemented our own custom user-level context management routines. Our approach to saving and restoring contexts was inspired by the Hood user-level threading library (Papadopoulos, 1998). To suspend a context, the runtime saves all user-level registers on top of the current stacklet, so that the stacklet is a self-contained record of the context state. To resume a context, the runtime restores registers from the stacklet, and then continues execution.

Figure 5.12 gives the prototypes of the three primary context manipulation

functions. The first function, `save_and_move_context`, is used as part of a `spawn` operation to save the continuation of the spawned method, set the stack pointer to point to a new stacklet, and then continue executing with the same register values. The first action of the function is to save all user registers to the top of the current stacklet. Invoking the `save_and_move_context` function causes the compiler to emit code to save all caller-saved registers, so this function only needs to save the remaining callee-saved registers. The function then saves the resulting stack pointer at the address pointed to by the `save_sp` argument. Finally, `save_and_move_context` sets the stack pointer to the value specified by the `new_sp` argument, which switches to the new stacklet.

The `save_and_move_context` function forks a single context into two contexts, one using a new stacklet, and the original context, which keeps using the original stacklet. Like the `fork` system call in UNIX, these contexts are differentiated by the return value of the function. A return value of zero indicates the program is executing in the new context, and a non-zero return value indicates the original context.

The second function, `restore_context`, resumes a suspended context after completion of a spawned task. This function switches the stack pointer to the value specified by `new_sp`, which points to an existing stack associated with a suspended context. Next, this function restores the user registers of the context from the top of the stack. Finally, `restore_context` returns the value specified by `ret_val`, which is always non-zero, indicating the resumption of a saved context. When `restore_context` returns, it transfers control to the continuation of the function that saved this context (for example, `save_and_move_context`), and thus the return value appears to be the return value of that function.

The third function, `save_and_restore_context`, is used by the scheduler when switching from its context to a stolen continuation. This function combines the functionality of `save_and_move_context` and `restore_context` to suspend one context and resume another suspended context.

The context manipulation functions are implemented in assembly language for each of the platforms that PROMETHEUS supports. We originally used the context manipulation functions for 32-bit x86 and SPARC-V9 from the Hood library (Papadopoulos, 1999). We then rewrote these routines from scratch, significantly reducing the number of instructions necessary for each operation, and then ported the resulting functions to 64-bit x86 and SPARC-V9. We note that the x86 functions are very short (16 instructions or less), due to the small number of registers that must be saved and restored. The SPARC-V9 functions are slightly longer, due to the large number of registers in the SPARC-V9 ISA, and the need to flush contents of

the register windows to the stack any time a context is saved.

Work-Stealing Deques

As described in Section 5.1, a work-stealing deque serves as the work queue for each worker thread. Note that a work-stealing deque is not a general concurrent deque, which allows concurrent operations on both ends of the queue, such as the implementation described by Herlihy et al. (2003). Instead, a work-stealing deque is accessed by a single thread at one end (the *bottom*), and may be accessed by multiple threads at the other end (the *top*). For the sake of brevity, we will refer to work-stealing deques simply as deques henceforth.

Mohr et al. (1991) introduced the use of deques in work-stealing schedulers. Frigo et al. (1998) showed how to eliminate the need to lock the deque for most local accesses to the deque by its associated worker thread. They use a protocol called THE, similar to that proposed by Dijkstra (1983), which allows accesses to the bottom of the deque to avoid locking, unless there is exactly one item in the deque. Arora et al. (1998) improved on this design further with a non-blocking deque that ensures the progress condition of lock-freedom. This design allows for most accesses to the bottom of the deque (again, unless the deque contains exactly one element) to be performed with ordinary loads and stores.

All of the deque implementations we have described so far use a fixed-size array as storage for the elements in the deque. If the deque size chosen for a particular run of the program is not large enough, it will overflow and cause the program to crash. Chase and Lev (2005) proposed a deque that replaces the array of the Arora et al. deque with a dynamically resizable circular array. This design maintains the lock-freedom of the Arora et al. design, while allowing the deque to automatically grow when an insertion into the deque would cause it to overflow. The Chase and Lev deque is currently the state-of-the-art design for work-stealing deques, and is used as part of the Java Fork/Join framework (Lea, 2000). We adopted this design without modification for the PROMETHEUS system.

We note that while the deque implementations due to Frigo et al. (1998), Arora et al. (1998), and Chase and Lev (2005) all promise to synchronize most accesses to the bottom of the deque with ordinary loads and stores, this is infeasible on real machines. Any machine that does not implement sequential consistency requires that a memory fence be placed after accesses to the bottom of the deque. Michael et al. (2009) observe that on many processors, using an atomic instruction to adjust the bottom pointer may be as fast or faster than the combination of ordinary load and stores plus a memory fence.

```
1  template <typename T>
2  class ws_deque_t {
3  public:
4      ws_deque_t (int initial_size);
5      void push_bottom (T* item);
6      T* pop_bottom ();
7      T* steal ();
8  };
```

Figure 5.13: Work-stealing deque interface

Figure 5.13 gives the interface for the work-stealing deque. The class is implemented as a template, and may be specialized to allow it to be used for tasks with eager task creation, or continuations with lazy task creation. The constructor (line 4) initializes the deque to an initial size as specified by its sole parameter. A worker thread uses the `push_bottom` (line 5) and `pop_bottom` (line 6) methods to push and pop work from its local deque. Workers use the `steal` method when attempting to steal work from another thread.

5.5 RUNTIME OPERATIONS

In this section, we describe how the PROMETHEUS runtime uses the data structures described in the previous section to implement dynamic scheduling via work stealing.

Initialization

When the PROMETHEUS runtime library is initialized, it performs a series of actions to prepare to execute a task-based program. First, the runtime spawns a number of worker threads. By default, the total number of threads is set to be equal to the number of hardware contexts in the system, but this value may be overridden at run time by setting an environment variable. Second, the runtime constructs a work-stealing deque for each thread. Third, the runtime creates a *scheduling context* for each worker, which encapsulates the register and stack state of the scheduling code for each worker thread. Then the initialization routine returns and the program continues execution.

Spawning and Synchronizing Tasks

Now that we have described the primary data structures used in the PROMETHEUS runtime, we can put all the pieces together to show how the system schedules tasks. When a worker thread is executing program code and encounters a `spawn`, it performs the following actions:

1. Allocate a new stacklet.
2. Save the current context, and change to a new context using the new stacklet.
3. Push the previous context, which contains the continuation of the spawn, into the local deque of the worker thread.
4. Execute the task.
5. Switch to the scheduling context to obtain more work.

We now review these steps in detail. Figure 5.14 lists the code for spawning tasks, which is broken into two functions: the `spawn` function (lines 1–17) called by the programmer, and the `execute_task` function (lines 19–33) used by the runtime to execute the task. The `spawn` function begins by determining the ID of the current thread (line 2), and then accessing the corresponding context to add one to the task count of the termination-detecting barrier (line 3). (Recall that this count must be incremented before the spawn of the task to ensure correct operation of the barrier.) The `spawn` function then copies the task pointer into a global array at the index corresponding to the current thread ID (line 4). This is necessary because the local variables of the currently executing function will become inaccessible when the PROMETHEUS context-switching function changes the stack pointer to a new stacklet.

Next, `spawn` prepares to switch to a new context in which it will execute the task. First, it determines the address of the location in the context structure where the stack pointer will be saved and copies it into a local variable (line 6). Second, it allocates a new stacklet (line 7), and third, it determines the initial stack pointer of the new stacklet (line 8). The `spawn` function then changes contexts by calling `save_and_move_context` (line 11). After saving the registers of the current context on the stack, this function changes the stack pointer to the newly allocated stack and returns zero. Then `spawn` function then calls `execute_task` (line 12) to execute the task in the new context. This function does not return, and thus this new context does not execute the rest of the code in `spawn`. When the saved context is eventually restored—either by the current thread, after the spawned task completes, or by a thread that steals the saved context—the `save_and_move_context`

```
1  void spawn (task_t* task) {
2      tid_t tid = get_thread_id ();
3      context[tid]->td_barrier->add_tasks (1);
4      tasks[tid] = task;
5
6      void** save_sp = &context->save_sp;
7      stacklet_t* stacklet = allocate_stacklet ();
8      void* new_sp = stacklet->get_sp ();
9
10     // fork point, return value of zero indicates new context
11     if (save_and_move_context (save_sp, new_sp) == 0) {
12         execute_task (); // does not return
13     }
14
15     // if return value not zero, executing continuation
16     return;
17 }
18
19 void execute_task (void) {
20     tid_t tid = get_thread_id ();
21     context_t* continuation = context[tid];
22     td_barrier_t* td_barrier = continuation->td_barrier;
23     deque[tid].push_bottom (continuation);
24
25     task_t* task = tasks[tid];
26     task->execute ();
27     td_barrier->subtract_tasks (1);
28     delete task;
29
30     // thread id of current worker may have changed, must re-read
31     tid = get_thread_id ();
32     restore_context (scheduler[tid], (void*) 1);
33 }
```

Figure 5.14: Implementation of spawn

```
1  void sync () {
2      tid_t tid = get_thread_id ();
3      td_barrier_t* td_barrier = context[tid]->td_barrier;
4      while (!td_barrier->complete ()) {
5          yield ();
6      }
7  }
```

Figure 5.15: Implementation of `sync`

function (line 11) will return a non-zero value, and execution will resume at line 21. At this point, the spawning function is complete and the program continues.

The `execute_task` function (lines 19–33) is responsible for executing the task in the new context. First, it retrieves the previous context (line 21) and pushes it into the bottom of the local deque (line 23). This context, which contains the continuation of the task, will either be resumed by the current thread when the task is complete, or stolen by another thread that runs out of work. Next, `execute_task` retrieves the task from the worker thread's slot in the task array (line 25) and executes it (line 26). Once the task completes, the task count of the termination barrier is decremented by one (line 27), and the task object is deallocated (line 28).

When it finishes executing the task, `execute_task` must return to the scheduling context, which will acquire more work. However, it is possible that this code executes in a different worker thread than the beginning of the function. This can occur when the executed task spawns one or more nested tasks, which could result in the continuation of a nested task (including previous invocations of `execute_task`) being stolen by another worker. It is therefore necessary to re-read the thread ID (line 31) to ensure that control is transferred to the scheduling context of the current worker thread (line 32). The scheduling context resumes the scheduling loop, which looks for work in the local deque, and then attempts to steal work when the local deque becomes empty.

Figure 5.15 lists the PROMETHEUS implementation of the `sync` primitive, which causes the worker thread in which it is invoked to wait until all outstanding tasks complete. This function identifies the termination-detecting barrier of the current context (line 3), and loops until the `complete` method of the barrier returns true. To ensure good multi-tasking performance, `sync` follows the recommendation of Arora et al. (1998), yielding the processor between each unsuccessful call to `complete`.

```
1  void schedule () {
2      tid_t tid = get_thread_id ();
3      context_t* continuation = deque[tid].pop_bottom ();
4
5      // loop until terminated
6      while (true) {
7
8          // while continuation in local deque, pop and execute
9          while (continuation != NULL) {
10             void** save_sp = get_current_sp (tid);
11             void* sp = continuation->get_sp ();
12             save_and_restore_context (save_sp, sp, 1);
13             // control transferred to continuation
14
15             // return from continuation
16             recycle_stacklet (tid);
17             continuation = deque[tid].pop_bottom ();
18         }
19
20         // no continuation in local deque
21         while (continuation == NULL) {
22             check_for_termination ();
23             thread_t::yield ();
24             tid_t victim = get_victim (tid);
25             continuation = deque[victim].steal ();
26         }
27     }
28 }
```

Figure 5.16: Scheduling loop

Scheduling Loop

Workers acquire work by entering the scheduling context, which continuously executes the *scheduling loop* listed in Figure 5.16. When the scheduling context is active, this loop (lines 6–27) runs continuously until it finds work or the runtime is terminated. The scheduling loop comprises two inner loops. The first loop executes as long as the local deque has work (lines 9–18). It attempts to pop work from the deque 17, and if it is successful, it suspends the scheduling context and restores the continuation stored in the context popped from the deque (lines 10–12). When this work is completed, the scheduling context resumes at line 15. It

recycles the previous stacklet and then continues looking for work in the local deque.

Should the scheduling loop exhaust work in its local deque, it enters the second inner loop (lines 21–26). This loop first checks for termination (line 22). If the program has not signalled termination, the loop then yields the processor (line 23), allowing other workers with a chance to run on its processor. After yielding, it identifies a victim thread at random (line 24) and attempts to steal work from this worker (line 25). Once work is successfully stolen, the second loop exits and the outer loop then re-enters the first loop, once again executing work from the local queue.

The `spawn` and `sync` operations and the scheduling loop act in concert to dynamically schedule the tasks submitted to the runtime. These actions are performed by worker threads operating independently on decentralized work queues. Coordination between worker threads only occurs when a worker exhausts its local work and attempts to acquire additional work from another worker. The decoupled nature of the work-stealing algorithm thus leads to an elegant and efficient means for distributing work to the hardware contexts of a multiprocessor system.

5.6 SUMMARY

In this chapter, we described the PROMETHEUS runtime system for dynamic task scheduling. We began by reviewing dynamic scheduling algorithms, and summarized previous work that established the time- and space-efficiency of work stealing. We then described two key aspects of the design of a work-stealing scheduler: creating tasks, which may be performed eagerly or lazily; and managing the tree structure task activation records, which map awkwardly onto the linear stack structure used by sequential languages. PROMETHEUS improves upon previous work-stealing algorithms by implementing lazy task creation, which preserves the space-efficiency guarantee of the scheduling algorithm; and by using a combination of fast user-level context switching and efficient allocation of stacklets to provide independent storage for activation records of independent tasks. These improvements enable PROMETHEUS to realize the full potential of work-stealing algorithms, which has previously required compiler support, while maintaining full compatibility with existing C++ programs, and requiring no additional effort from the programmer. After describing the high-level design decisions, we described the implementation of the data structures and operations of the runtime system. In conclusion, the PROMETHEUS runtime enables data-driven decomposition of

C++ programs using the task-based serializer described in Chapter 4 to exploit state-of-the-art dynamic scheduling.

6 RECEIVER DISAMBIGUATION AND PARALLEL LOOPS

> *There are known knowns. There are things we know that we know. There are known unknowns. That is to say there are things that we now know we don't know. But there are also unknown unknowns. There are things we do not know we don't know. ... It sounds like a riddle. It isn't a riddle. It is a very serious, important matter.*
>
> — Donald Rumsfeld (2002)

Data-driven decomposition achieves repeatable and predictable parallel execution by invoking methods on each private object in the same order as the sequential program. To preserve this ordering, each parallel operation identifies its *receiver*—the object it will modify—and the runtime ensures that the operation is ordered after earlier operations on that object. Therefore a particular operation cannot modify its receiver until the identities of the receivers of all operations that occur earlier in the sequential ordering are known. The *receiver identification problem* is a fundamental limitation of any parallel execution model that dynamically preserves a program-specified ordering of operations on each data element in the program.

The implementation of data-driven decomposition we have described in earlier chapters of this dissertation relies on *explicit delegation*, which requires each operation to identify its receiver before commencing parallel execution. Performing receiver identification sequentially ensures that all previous operations have named their receivers, so the operation may immediately delegate a method invocation to the specified receiver. However, if each operation spends a significant fraction of its execution time identifying the data it will modify, receiver identification may become the critical path through the program, limiting its parallelism.

In this chapter, we introduce *ambiguous delegation* to overcome the potential sequential bottleneck of explicit delegation. Ambiguous delegation allows an operation to immediately begin parallel execution. Initially, the operation performs the computation needed to identify the object it will modify. During this phase, the receiver is *ambiguous*. Once the operation identifies its receiver, this ambiguity is *resolved*. The operation then performs *receiver disambiguation* to ensure that delegation of a method invocation to the serializer of its receiver does not occur until it is certain that any earlier delegation that may name the same receiver is complete. Once the receiver is disambiguated with respect to earlier operations, the operation then delegates a method invocation to the serializer of its receiver,

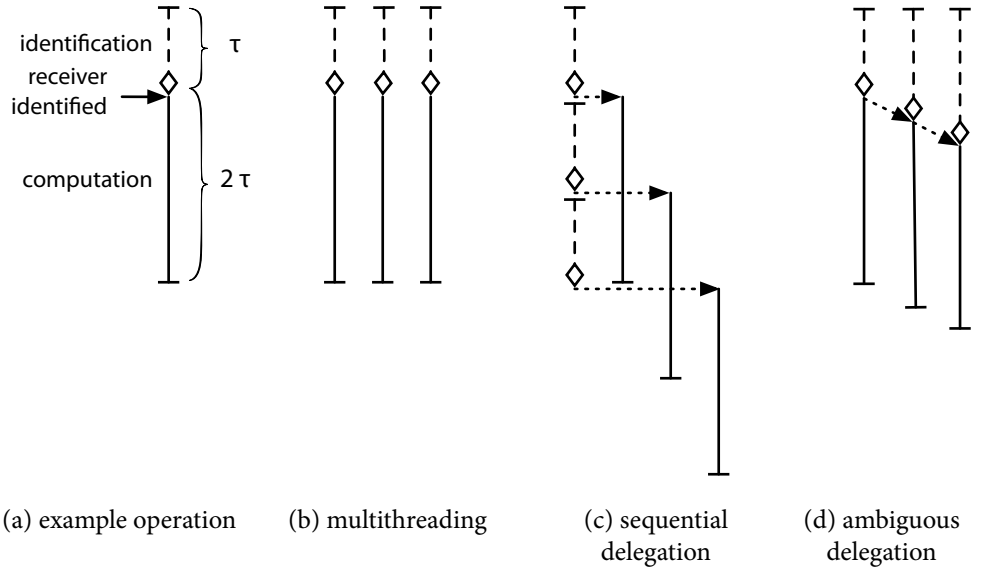
which will order the operation after any earlier method invocations delegated to that serializer. We describe the receiver identification problem and receiver disambiguation in more detail in Section 6.1.

As an initial case study, we apply ambiguous delegation to the implementation of efficient parallel loops. Many parallel execution models provide looping constructs that allow the programmer to indicate that each loop iteration operates on independent data. The runtime system then executes these loops in parallel by scheduling loop iterations on a set of worker threads. To minimize the impact of scheduling on the critical path, efficient parallel loop implementations use a divide-and-conquer strategy to parallelize the assignment of loop iterations to worker threads. Parallelizing both the distribution and execution of loop iterations significantly reduces the granularity of work required to profitably parallelize a loop. In Section 6.2, we review divide-and-conquer parallel loops, and show how the receiver identification problem impedes parallelizing loops in data-driven decomposition. Then in Section 6.3, we propose the *receiver disambiguation queue* (RDQ) to facilitate ambiguous delegation. Finally, in Section 6.4, we describe how to use the RDQ to implement an efficient parallel `foreach` loop construct for data-driven decomposition.

6.1 THE RECEIVER IDENTIFICATION PROBLEM AND AMBIGUOUS DELEGATION

Data-driven decomposition dynamically parallelizes method invocations that operate on disjoint objects. Frequently, the identities of these objects are readily available in the program—accessing an object may be as simple as dereferencing a pointer or indexing into an array. However, some algorithms require more substantial preliminary processing to identify the receiver object. This can significantly hamper the parallelism of a program performing explicit delegation, since all processing that occurs before the receiver is identified will impact the critical path of the program. To illustrate this issue, we view each computational operation as comprising two phases: First, the *receiver identification phase* determines the object the operation will modify. Second, the *computation phase* invokes a method on the object named by the identification phase.

Figure 6.1 illustrates several scenarios for parallel execution of such computations. For the purposes of this example, we assume each operation is composed of an identification phase (represented as a dotted line) that takes time τ and a computation phase (represented as a solid line) that takes time 2τ, as shown in Figure 6.1(a). The point in the operation where the identification phase completes and names the receiver is shown as a diamond. We further assume that the partic-



(a) example operation (b) multithreading (c) sequential delegation (d) ambiguous delegation

Figure 6.1: Illustration of the receiver identification problem

ular operations shown in the figure are independent and may execute in parallel, and we ignore any scheduling overheads.

Figure 6.1(b) depicts these operations executing as a multithreaded program. Because multithreading permits shared data, the computation phase would use mutual exclusion when necessary to ensure the atomicity of its calculations. As we described in Chapter 2, multithreaded programs sacrifice repeatability and predictability because a particular object may be manipulated by operations in different threads with no restriction on how these operations are ordered. However, for this example, multithreading yields high performance because it parallelizes both the identification and computation phases of each operation.

Figure 6.1(c) shows a data-driven decomposition of the same set of operations using explicit delegation. The receiver identification phase of each computation executes sequentially to ensure that method invocations are delegated to a particular serializer in program order. Once the receiver is named, the program delegates the computation phase. Assuming the identification phase is sequential and the computation phase is fully parallel, Amdahl's Law (Amdahl, 1967) dictates that the upper limit of speedup of computations that spend a fraction f_{id} in the

identification phase is:

$$\text{Speedup}_{\infty} = \frac{1}{f_{id}} \tag{6.1}$$

on an infinite number of processors. For the example shown in Figure 6.1, the identification phase constitutes one-third of the operation, and the computation phase makes up the remaining two-thirds. According to Amdahl's Law, this computation can achieve a maximum speedup of three, even on an infinite number of processors.

Figure 6.1(d) illustrates how we may alleviate the receiver identification problem with ambiguous delegation, which relaxes the requirement that an operation names its receiver before commencing parallel execution. Ambiguous delegation overlaps the receiver identification phases of multiple operations, but maintains the sequential order that method invocations are delegated to the serializer of a particular object. Once an operation names its receiver, it performs receiver disambiguation, which stalls the operation until all previous operations that might identify the same receiver complete their receiver identification phase and delegate a method invocation to the serializer of the identified object. Only then can the operation proceed to delegate a method invocation to the designated receiver. In Figure 6.1(d), the sequential ordering enforced by receiver disambiguation is indicated by the arrows between the diamonds at the end of each receiver identification phase. Enforcing this ordering may impact performance when a short identification phase must wait on an earlier, longer identification phase. However, only this imbalance contributes to the critical path of the program, in contrast with explicit delegation, where the entire identification phase is on the critical path.

Receiver disambiguation is analogous to *memory disambiguation* in a dynamically scheduled microprocessor. Memory disambiguation enforces the sequential semantics of load and store instructions that execute out-of-order by ensuring that a dynamic load instruction to a particular address always reads the value written by the most recent dynamic store instruction to that same address. The IBM System/360 Model 91 (Anderson et al., 1967) is an early example of a dynamically scheduled processor. This machine uses a fully associative *store queue* to maintain information about all in-flight store instructions (Boland et al., 1967). When a load issues, it calculates its address, and then searches the store queue to determine if there are any stores that it may depend on. If the store queue contains stores that have not yet generated their address, the load must stall, since one or more of these stores might be to the same address as the load. If the store queue contains

stores to the same address as the load, the load must stall until this store completes to ensure that the load reads the correct value from memory.

In this chapter, we propose a construct called the *receiver disambiguation queue* (RDQ), which performs the same function for receiver disambiguation that the store queue performs for memory disambiguation. In this analogy, a parallel operation identifying its receiver object corresponds to a memory instruction generating its address. The RDQ detects dependences between ambiguous delegations, just as the store queue detects dependences between in-flight memory instructions. After an ambiguous delegation identifies its receiver, it accesses the RDQ, which checks two conditions and takes appropriate action. First, if there is any earlier ambiguous delegation that has not identified its receiver, and if so, the operation stalls until this ambiguity is resolved. Second, if an earlier operation has identified the same receiver as the present operation, it stalls the present operation until the previous one completes delegation to the serializer of this receiver. Note that despite this strong conceptual analogy between receiver disambiguation and memory disambiguation, the RDQ is implemented in software and is thus necessarily different from a hardware structure like the store queue. We will describe the operation and implementation of the RDQ in Section 6.3.

While early dynamically-scheduled processors stalled loads when a an earlier store has not generated its address, more recent processors, such as the MIPS R10000 (Yeager, 1996), use speculative execution to allow load instructions to execute immediately, squashing and rolling back if the processor discovers the load has violated a dependence. Processors may also use dependence prediction techniques (Moshovos et al., 1997) to selectively execute load instructions speculatively when they are unlikely to cause a dependence violation. While we will not examine this issue in this dissertation, the receiver identification problem presents a similar opportunity to apply speculation. Instead of stalling, an operation that has named its receiver object, but is not the oldest ambiguous delegation, could begin executing speculatively. If an older ambiguous delegation identifies the same receiver object, the speculative operation would be squashed. If all previous ambiguous delegations identify different receivers, then the speculative execution could be allowed to commit, having avoided wasting time waiting on earlier operations.

While we address the issue of receiver identification in the context of data-driven decomposition, we note that this is a fundamental limitation of any parallel execution model that dynamically guarantees a program-specified ordering of operations on every variable or data structure. Receiver disambiguation overcomes this limitation and allows independent operations to perform both receiver iden-

tification and computation phases of a operation in parallel. For the remainder of this chapter, we will focus on the application of receiver disambiguation to divide-and-conquer parallel looping constructs. However, we believe that receiver disambiguation may prove to be useful in other situations, such as computations that perform lookups in complicated container structures to identify the object on which to perform an operation.

6.2 EFFICIENT PARALLEL LOOPS

Many parallel execution models provide parallel looping constructs, which are a straightforward and idiomatic way to express parallel computation. The `parallel_for` loop in OpenMP, Cilk++, and TBB and the `DOALL` loop in parallel FORTRAN are examples of such constructs. To achieve parallel execution, a parallel loop must distribute its work (loop iterations) to worker threads for execution. Figure 6.2 shows the DAG models (as described in Section 5.1) for two possible approaches to executing a parallel loop with eight iterations. The first approach, shown in Figure 6.2(a), sequentially spawns each loop iteration as a task. Recall that the span of a computation is determined by the critical path through its execution DAG, illustrated by the gray nodes in the figure. If we assume each loop iteration performs constant work, i.e., $T_1 = O(1)$, then the span of this computation is linear in the number of iterations:

$$T_\infty = O(n_{\text{iterations}}). \tag{6.2}$$

The problem with this strategy is that it limits the parallelism of the loop—doubling the number of loop iterations doubles the span of the computation. While it may provide adequate performance when the loop iterations perform a large amount of work, the span of the loop will become the limiting factor for shorter loop iterations.

To reduce the span, we must parallelize not only the loop iterations, but the distribution of loop iterations to threads. Figure 6.2 illustrates a more efficient parallel loop implementation that recursively traverses the loop range in a divide-and-conquer fashion, spawning tasks for each loop iteration at the leaf nodes of the recursion. Note that all paths through this DAG are the same length, and one of them is highlighted in gray to illustrate the critical path. Using the divide-and-conquer approach, doubling the number of loop iterations increases the span by

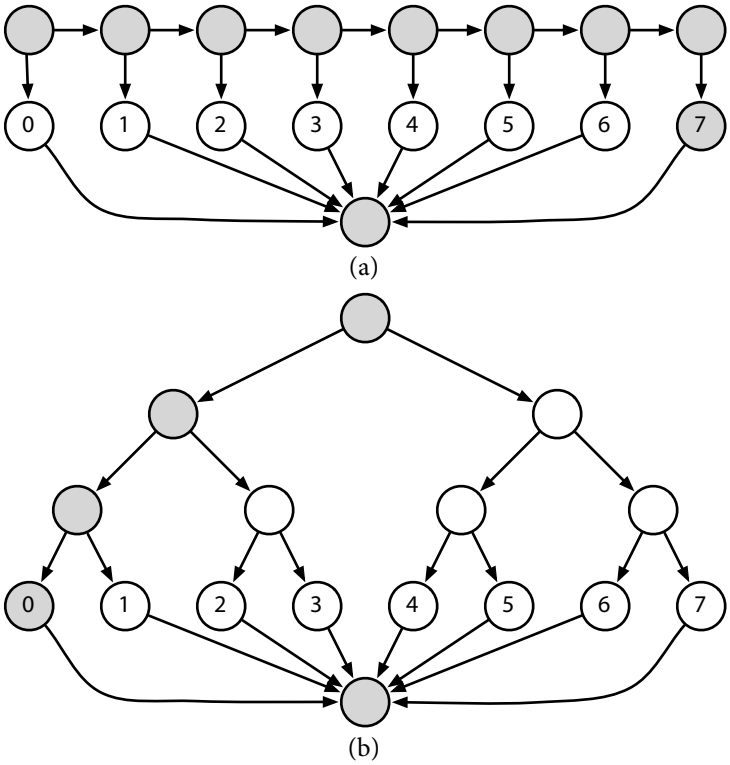


Figure 6.2: A DAG model of a parallel loop using (a) sequential and (b) divide-and-conquer distribution of loop iterations

one, and thus the span is logarithmic in the number of iterations:

$$T_\infty = O(\lg n_{\text{iterations}}). \tag{6.3}$$

Because of the improved parallelism provided by the divide-and-conquer strategy, tasking systems such as Cilk++ (Leiserson, 2009) and TBB (Kukanov and Voss, 2007) implement parallel loops in this way.

PROMETHEUS supports parallel loops via the `foreach` construct using the interface given in Figure 6.3. This construct implements a parallel loop by invoking a method on all elements of a C++ STL container in the half-open range `[begin,end)`.[1] The programmer uses `foreach` to indicate that all objects in this

[1] A half-open range `[begin,end)` includes the elements `begin`, `begin+1`, ..., `end-1` (but not `end`). Half-open ranges are commonly used for iterators because they simplify determining the

```
1  namespace prometheus {
2       // Parallel loop over a half-open range [begin, end)
3      template <typename T, typename C, typename B, typename Args... args>
4      void foreach (typename T <C>::iterator begin,
5                    typename T <C>::iterator end,
6                    void (B::*method) (Args...),
7                    Args... args);
8
9      // Parallel loop over a half-open range [begin, end)
10     // Each task is N loop iterations specified by grain_size
11     template <typename T, typename C, typename B, typename Args... args>
12     void foreach (size_t grain_size,
13                   typename T <C>::iterator begin,
14                   typename T <C>::iterator end,
15                   void (B::*method) (Args...),
16                   Args... args);
17 }
```

Figure 6.3: PROMETHEUS API for parallel loops

range are distinct, which allows the runtime to optimize the parallel loop. The sequential elision of the `foreach` loop is an ordinary `for` loop that invokes the method on each object in the range. The parallel execution of `foreach` provides ensures equivalent semantics.

The `foreach` function is parameterized on three types: the type `T` of the C++ STL container, the class `C` of the objects stored in the container, and the class `B` in which the method to be called is declared. There are two versions of the `foreach` function: the first version, which forms tasks of individual loop iterations (lines 3–7); and the second version, which has an additional `grain_size` that specifies the number of loop iterations used to form a task (lines 11–16). The other parameters to `foreach` are iterators designating the beginning and end of the half-open range (`begin` and `end`), a C++ method pointer to the method to invoke on each object in the range (`method`), and a list of arguments to pass to the method (`args`).

Using explicit delegation, the receiver identification problem introduces a trade-off in the implementation of `foreach`. To illustrate this problem, we present the code in Figure 6.4. This snippet is a function with a single parameter that takes a C++ STL `vector` of objects by reference. (A `vector` is an array container.) The

size of the range, which is given by `end-begin`, and detecting iteration termination, which is indicated when the iterator is equal to `end`.

```
1  void do_foo (vector<obj_t>& array) {
2      foreach (array.begin (), array.end (), &obj_t::foo);
3      array[0].delegate (&obj_t::bar);
4  }
```

Figure 6.4: Example of the receiver identification problem with a parallel loop

function then uses the PROMETHEUS `foreach` construct to perform a parallel loop that invokes the method `foo` on each element in the vector (line 2). Finally, the function delegates an invocation of the method `bar` on the first element in the vector (line 3).

If `foreach` is implemented using the naive approach to the distribution of loop iterations shown in Figure 6.2(a), this function will work correctly, because method invocations will be delegated sequentially. However, this implementation of `foreach` yields no performance advantage over using a normal `for` loop that delegates the method on each element of the vector.

To achieve a higher-performing parallel loop, delegation of the method invocations in the loop must proceed in parallel, as shown in Figure 6.2(b). Because the programmer ensures that the elements in the vector are disjoint objects, a `foreach` implementation that delegates the method invocations in parallel will not introduce races when inserting method invocations into the respective serialization queues of each object. The problem occurs when the program reaches the delegation of `bar` on line 3. Because this version of `foreach` performs delegations in parallel, the delegation of `foo` on `array[0]` may occur in a different worker thread than the delegation of `bar` on the same object. This introduces a determinacy race between the two delegations as they attempt to insert a method invocation into the serialization queue of `array[0]`. A straightforward solution to this problem is to use a termination-detecting barrier to ensure that all ambiguous delegations in the `foreach` loop complete before the program continues. This approach has its own disadvantage—it prevents parallelism between the loop and the subsequent code, because it requires the delegation of `bar` on `array[0]` to wait for all delegations in the loop to complete, rather than just the delegation on `array[0]`.

The need to preserve the ordering of delegations that may occur in different worker threads is the source of the trade-off between loop efficiency and parallelism. This is exactly the situation that ambiguous delegation addresses. Because the programmer specifies that the each iteration manipulates a distinct object,

the runtime may schedule them using the divide-and-conquer strategy without violating the sequential ordering. The operations in the loop can then use receiver disambiguation to ensure that a particular loop iteration is ordered correctly with respect to operations before and after the loop, without resorting to barriers. In the next section we will describe the operation and implementation of the receiver disambiguation queue, which preserves the correct ordering of method invocations on an object in the presence of ambiguous delegation.

6.3 THE RECEIVER DISAMBIGUATION QUEUE

PROMETHEUS supports ambiguous delegation using the receiver disambiguation queue (RDQ). This RDQ removes the requirement that an operation explicitly identify its receiver before commencing parallel execution. It does this by conservatively stalling the computation phase of any operation until it can determine that all previous operations to that receiver have completed delegation, or that there are no longer any previous operations with ambiguous receivers.

The RDQ contains a list of nodes, each of which represents a set of operations. This set may include either a single operation or multiple operations to disjoint objects, such as a parallel loop. The RDQ disambiguates receivers between an operation and all operations represented by earlier nodes. It does not disambiguate operations represented by the same node, which are assumed to be independent. This allows a loop to schedule its iterations in any order, facilitating divide-and-conquer parallelization.

Each RDQ node maintains a count of the number of operations it represents, and a timestamp indicating the ordering of this set of operations relative to earlier and later operations in the sequential execution. Delegations, whether explicit or ambiguous, records the node at the tail of the RDQ, which we refer to as the *disambiguation node* of that operation. The disambiguation node summarizes the state of all previous ambiguous delegations. After recording this node, the ambiguous delegation adds a new node to the tail of the list, which we refer to as the *ambiguity node* for that operation. The ambiguity node tracks the completion of the delegations associated with it, and is used for ordering later operations. Once a particular operation identifies its receiver and the associated serializer, it disambiguates this serializer with the disambiguation node, which stalls the operation until any previous parallel operations that may identify the same receiver have completed delegation. When disambiguation completes, the operation delegates a method invocation to the specified serializer.

The function of the RDQ is straightforward, but its operation is complex. We

```
1  // declare array of 3 elements and initialize
2  vector <object_t> array (3);
3  initialize_array (array);
4
5  foreach (array.begin (), array.end (), &object_t::A);
6  foreach (array.begin (), array.end (), &object_t::B);
7  foreach (array.begin (), array.end (), &object_t::C);
8  array[0].delegate (&object_t::D);
9  array[1].delegate (&object_t::E);
10 array[2].delegate (&object_t::F);
```

Figure 6.5: Code for RDQ example

now present an extended example to illustrate how the RDQ disambiguates the receivers of ambiguous delegations.

An Example of RDQ Operation

We illustrate the operation of the RDQ using the example code in Figure 6.5. This program first creates and initializes an array of three elements (lines 2–3). It then executes three `foreach` loops, which each apply three methods to each element in the array: `A` (line 5), `B` (line 6), and `C` (line 7). The program then performs explicit delegation of the methods `D`, `E`, and `F` on the first, second, and third elements of the array, respectively (lines 8–9).

Figures 6.6 and 6.7 depict one possible execution of this program. The RDQ is shown on the left, and the method invocations enqueued in the serializer of each element of the array are shown on the right. The subscript of each method invocation indicates which array element to which that invocation will be applied.

The RDQ tracks each set of operations with a list node that records the timestamp, a counter for the number of operations, and a boolean value indicating whether the ambiguities represented by the node have been resolved. New nodes are appended to the tail of the list. The RDQ maintains the invariant that the head always points to a resolved node. Initially, both the head and tail of the list point to a dummy node with a counter set to zero that is marked as resolved, as shown in Figure 6.6(a).

When the code listed in Figure 6.5 executes, the program creates and initializes the array, and then encounters the first `foreach` loop (line 5). Before the loop begins, it notifies the RDQ that it will perform three independent ambiguous

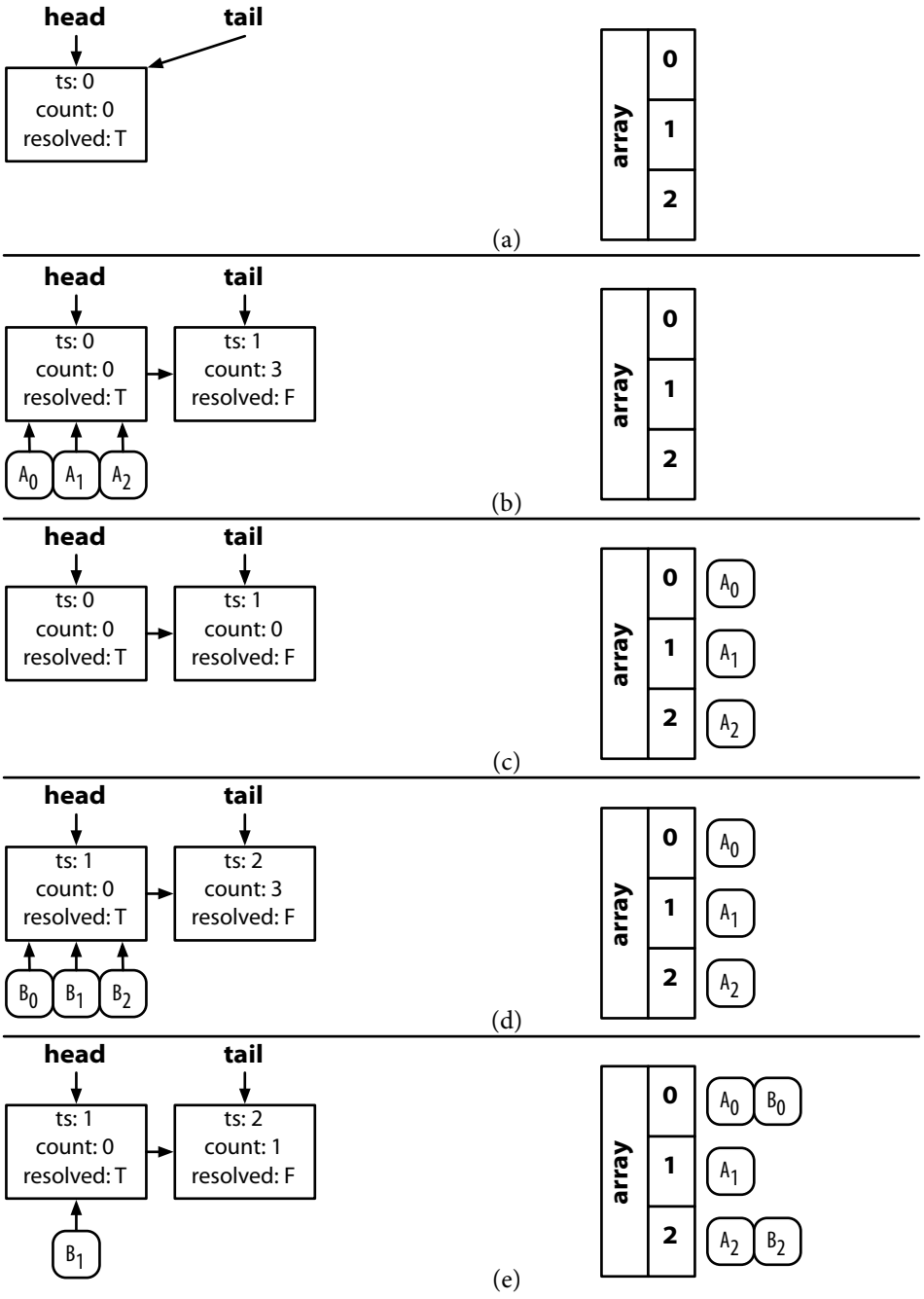


Figure 6.6: Example of RDQ operation

delegations. The RDQ records the disambiguation node at the tail of the list, which will be returned to the `foreach` loop to use for receiver disambiguation with earlier operations. Then the RDQ creates a new node, marking it with the timestamp of the previous node incremented by one, setting the number of operations to three, and initializing the resolved flag to false. It appends this new node to the tail of the list to reflect the loop of line 5.

Figure 6.6(b) shows the state of the program's execution just before the `foreach` loop on line 5 spawns tasks to execute each iteration. As the program executes the loop, it recursively spawns tasks that subdivide the loop range until it is reduced to a single element. The tasks at the leafs of this recursion are responsible for executing the individual loop iterations, so we refer to them as *iteration tasks*. Before the delegating the `A` method to the specified element of `array`, each iteration task disambiguates the serializer of this object using the disambiguation node returned by the RDQ. Disambiguation checks two conditions: First, it checks if the serializer has passed through all previous ambiguous delegations, indicated when its timestamp is equal to that of the disambiguation node, at which delegation may safely proceed. Second, it checks if the ambiguity of the RDQ node via the resolved flag, which indicates all previous ambiguous operations have completed. If neither of these conditions is satisfied, that particular loop iteration must stall and periodically recheck these conditions until one becomes true.

The timestamps of the serializers for the objects of `array` are all initially zero. Since this matches the timestamp of the RDQ node, the first disambiguation condition is satisfied. (Since the resolved flag of this node is set, the second condition would also be satisfied.) Thus the iteration tasks proceed to delegate the invocations of `A` to the serializers of the three array elements. Once an iteration task completes delegation, it notifies the RDQ that it has resolved an ambiguous operation, it decrements the counter of the ambiguity node (which is always the node after the disambiguation node) by one. Figure 6.6(c) shows the state of the RDQ when all loop iterations have delegated their method invocations and the counter of the RDQ node has been decremented to zero. The resolved flag is not yet set, because it is lazily updated by later operations.

The next step in the program is the second `foreach` loop on line 6 of Figure 6.5, which delegates `B` to each object in the array. The `foreach` loop informs the RDQ that it is initiating a set of three independent ambiguous delegations. Before recording this ambiguity, the RDQ *cleans* the list, which performs two functions: propagating the resolved flag through completed nodes, and reclaiming the storage for completed nodes. The cleaning process begins at the head of the list, and checks if the count of the next node is zero. If it is, it sets the resolved flag to true, points

the head pointer at this node, and pops off the old head. This procedure is repeated until a successor of the head with a counter greater than zero is found, leaving a single resolved node at the head of the list. After cleaning is complete, the RDQ then adds a new node at the tail of the list to reflect the ambiguity introduced by the `foreach` loop, as shown in Figure 6.6(d).

Next, the `foreach` loop for `B` recursively spawns tasks to execute the loop iterations. After each iteration task identifies its receiver, it disambiguates its serialize. Since the iteration tasks for the previous loop iteration have already identified their receivers and completed delegation, the `foreach` loop can immediately spawn tasks for the `B` loop. To show how the RDQ enforces the ordering of operations on each serializer, let us assume that the loop iteration operating on array element 1 is delayed. This could happen in many different ways—for example, if the worker thread executing that task were preempted, or if there are more tasks in the system than workers to execute them. Figure 6.6(e) depicts this scenario, showing that method `B` has been delegated to array elements 0 and 2, but not element 1. Thus the corresponding RDQ node still has a counter value of one, indicating that one operation has not yet named its receiver.

Program execution proceeds to the third `foreach` loop on line 7 of Figure 6.5, which delegates `C` to each object in the array. Once again, `foreach` loop informs the RDQ that it is initiating three independent, ambiguous operations. The RDQ attempts to clean the list, but there is nothing to modify, since there is only one resolved node at the head of the list. Next, another node with an operation count of three is appended to the RDQ list as shown in Figure 6.7(a). The `foreach` loop then spawns tasks to execute the loop iterations. The resulting iteration tasks identify and disambiguate their receivers. The iteration tasks for array elements zero and two find that the serializers have a timestamp of 2, indicating that these serializers have been resolved with respect to previous ambiguous delegations, and so these tasks can safely delegate `C` to their respective serializers. Meanwhile, the iteration task for array element 1 finds that its serializer has a timestamp of 1, indicating that it has not completed the previous ambiguous operation. Its disambiguation node is not resolved, so this previous ambiguous operation may identify the same receiver (and in fact it will), so this iteration task must stall until the previous ambiguity is resolved.

This state of affairs is shown in Figure 6.7(b). The `A`, `B`, and `C` methods have been delegated on array elements 0 and 2, but only `A` has been delegated on array element 1, and `C` is being stalled by the delay in the iteration task responsible for delegating `B`.

The next step in the program is the explicit delegation of method `D` to `array[0]`

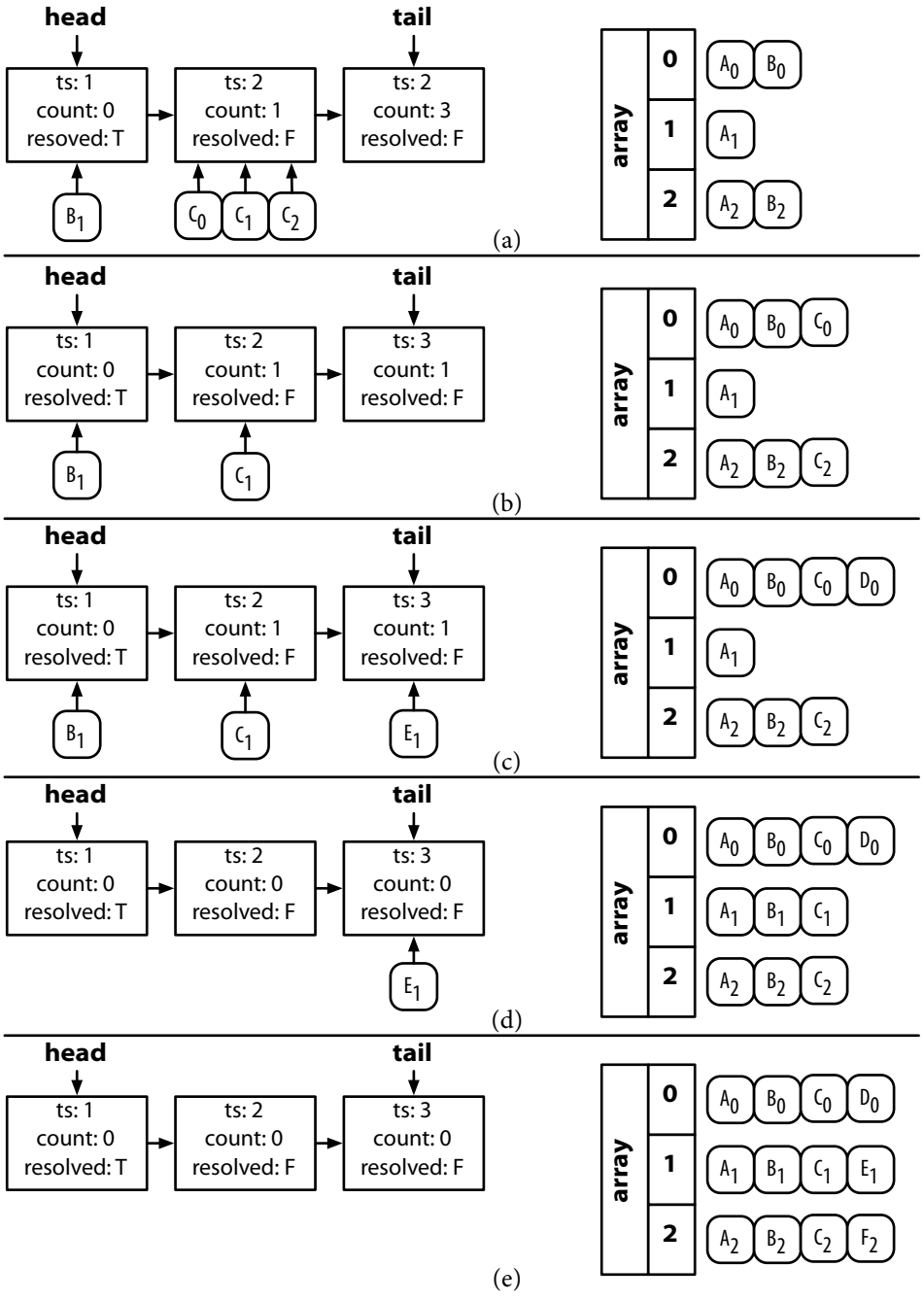


Figure 6.7: Example of RDQ operation, continued

(line 8 of Figure 6.5). Explicit delegation does not perform any modification to the RDQ, but it must disambiguate the serializer of the designated receiver with the RDQ to ensure the delegation is correctly ordered. The timestamp of the serializer for `array[0]` is 3, which matches the timestamp of the node at the tail of the RDQ, and thus this method may be immediately delegated. The next operation is the explicit delegation of method `E` on `array[1]` (line 9 of Figure 6.5). When this delegation performs disambiguation via the the RDQ, it finds that the timestamp of the serializer for `array[1]` is less than that of the node at the tail of the RDQ, and since this node is not flagged as resolved, the delegation stalls. Figure 6.7(c) shows the state of the program at this point. Since explicit delegation identifies receivers sequentially, program execution does not proceed past this point.

Now let us suppose the delay of the iteration task for the delegation of `B` on `array[1]` ends. After delegating `B` on `array[1]`, it increments the timestamp of the serializer of `array[1]`. When the task responsible for delegating `C` on `array[1]` rechecks this timestamp, it sees that the timestamp is now 2, which equals the timestamp of its disambiguation node, and so it proceeds to delegate `C` on `array[1]`, and then increment the timestamp of its serializer to 3. Figure 6.5(d) depicts this point in the program's execution.

Now that the timestamp of the serializer of `array[1]` is equal to 3, the explicit delegation of `E` on `array[2]` may proceed. Finally, the program reaches the explicit delegation of method `F` on `array[2]` (line 10 of Figure 6.5), and finds that its serializer's timestamp is 3, and since this is equal to the timestamp of the tail node of the RDQ, it immediately delegates the method. Figure 6.7(e) shows the final state of the program.

Note that the RDQ still contains several nodes and that the last two are not marked as resolved, although their counters have reached 0. The extra nodes will be removed by the cleaning routine, and the resolved flag will be set for these nodes. The cleaning routine is always executed when a new ambiguous operation begins, and is also periodically run by operations that are performing disambiguation in order to ensure that they see the latest values of the resolved flag.

This example illustrates how the RDQ performs receiver disambiguation to ensure ambiguous delegations maintain the sequential ordering of method invocations enqueued on a particular serializer. Even though the delegation of methods `C` and `E` on `array[1]` were ready to execute before the delegation of the method `B` on `array[1]`, the RDQ stalled delegation of these methods until after the delegation of `B` completed. Having demonstrated the operation of the RDQ, we now proceed to detail its implementation.

Implementation of the Receiver Disambiguation Queue

As we showed in the example, the RDQ relies on timestamps to discern when earlier ambiguous delegations have completed. This requires us to augment the serializer with a `time\-stamp` field, as well as an accessor method `get_time\-stamp` to read its value and a mutator method `set_time\-stamp` to set its value. For the sake of brevity, we do not show the code for these simple modifications here.

Figure 6.8 lists the code for the `rdq_node_t` class. Each instance of this class represents a set of ambiguous delegations to disjoint objects. The `rdq_node_t` class inherits from the `td_barrier_t` class (line 3) described in Section 5.4. Incorporating the functionality of the termination-detecting barrier provides an efficient counter to track the number of outstanding ambiguous delegations represented by the node.

The first field of the `rdq_node_t` class is a pointer to the RDQ containing this node (line 5). The second and third fields are pointers to the previous (line 6) and next (line 7) nodes in the list, respectively. The fourth field is the timestamp, which is implemented as a 64-bit unsigned integer (line 8). The fifth field is the `resolved` flag (line 9), which indicates that all ambiguous delegations for this node, as well as all previous nodes, have completed. This flag is a boolean value, augmented with a tag that is used during the cleaning process. We will explain the purpose of the tag when we describe the `clean` method of the RDQ.

The constructor of the RDQ node (lines 11–16) initializes the pointer to the RDQ and the timestamp using values specified by the parameters. The constructor also adds the number of ambiguous tasks specified by the `num_ambiguous` parameter to the counter contained in the `td_barrier_t` superclass (line 15).

Ambiguous delegations disambiguate their receiver using the serializer associated with the receiver and the node that was at the tail of the RDQ when the delegation began. This is done using the `disambiguate` method (lines 18–31). The single parameter to `disambiguate` is the serializer of the receiver named by the operation. This method stalls the operation in a loop (lines 19–30) that checks the two disambiguation conditions. First, it checks the serializer timestamp to see if it has completed all previous ambiguous operations (lines 21–22). Second, it checks if all previous ambiguous delegations have completed. Before performing this check, `disambiguate` invokes the `clean` method of the RDQ to ensure that the `resolved` flag has been propagated through the node list (line 25). Then it checks the `resolved` flag, and if it is set, then all previous operations have identified their receivers and completed delegation. When either of these two conditions is found to be true, the loop ends and the `disambiguate` method completes. As

```
1  typedef uint64_t ts_t;
2
3  class rdq_node_t : public td_barrier_t {
4  public:
5      rdq_t* rdq;
6      rdq_node_t* volatile prev;
7      rdq_node_t* volatile next;
8      ts_t timestamp;
9      tagged_bool_t resolved;
10
11     rdq_node_t (rdq_t* rdq, ts_t timestamp, size_t num_ambiguous) :
12         rdq (rdq), prev (NULL), next (NULL),
13         timestamp (timestamp), resolved (false)
14     {
15         add_tasks (num_ambiguous);
16     }
17
18     void disambiguate (const serializer_t* serializer) {
19         while (true) {
20             // check if serializer has passed through previous ambiguities
21             ts_t serializer_ts = serializer->get_timestamp ();
22             if (serializer_ts == timestamp) break;
23
24             // check if all previous ambiguities have been resolved
25             rdq->clean ();
26             if (resolved.value ()) break;
27
28             // yield the processor
29             yield ();
30         }
31     }
32
33     void resolve (serializer_t* serializer) {
34         serializer->set_timestamp (timestamp + 1);
35         next->subtract_task (1);
36     }
37 };
```

Figure 6.8: The RDQ node structure

```
1  class rdq_t {
2  private:
3      rdq_node_t* volatile head;
4      rdq_node_t* volatile tail;
5
6  public:
7      rdq_t () :
8          head (new rdq_node_t (this, 0, 0), tail (NULL))
9      {}
10
11     rdq_node_t* add_ambiguous (size_t num_ambiguous);
12     void disambiguate (const serializer_t* serializer) const;
13     void clean ();
14 };
```

Figure 6.9: The receiver disambiguation queue (RDQ) class

with all spin loops in the PROMETHEUS runtime, the loop yields the processor between loop iterations (line 29).

Upon completion, an ambiguous delegation calls the `resolve` method (lines 33–36). The single parameter is the serializer associated with the receiver named by the ambiguous delegation. The timestamp of this serializer is updated to match the timestamp of the node (line 34). The `resolve` method also increments the counter of the operation's ambiguity node (always the next node in the RDQ list after the operation's disambiguation node) (line 35). Recall that the disambiguation node represents all previous ambiguous computations, whereas the next node in the list represents the ambiguity of the operations associated with this node.

Figure 6.9 lists the code for the `rdq_t` class. The RDQ tracks the list of `rdq_node_t` objects with two fields that maintain pointers to the head and tail the list (lines 3–4). The constructor (lines 7–9 creates a dummy node with its count set to zero and sets the head and tail fields to point to this node. The `rdq_t` class has three other methods: the `add_ambiguous` method, which adds a node to the RDQ list, the `disambiguate` method, which disambiguates explicit operations with the RDQ, and the `clean` method, which propagates the `resolved` flag to completed nodes, and removes all but the most recently completed node.

The implementation of the `add_ambiguous` method is given in Figure 6.10. This method takes a single parameter that specifies the number of ambiguous delegations represented by this node. It first calls the `clean` method (line 3), and then allocates and initializes a new RDQ node and inserts it into the list (lines 6–9).

```
1  rdq_node_t* rdq_t::add_ambiguous (size_t num_ambiguous) {
2      // clean up completed nodes
3      clean ();
4
5      // allocate and initialize a new node
6      rdq_node_t* rdq_node = new rdq_node_t (this, timestamp, num_ambiguous);
7      rdq_node->prev = tail;
8      tail->next = rdq_node;
9      tail = rdq_node;
10
11     // return previous node for disambiguation
12     return rdq_node->prev;
13 }
```

Figure 6.10: Implementation of the `add_ambiguous` method of the RDQ

```
1  void rdq_t::disambiguate (const serializer_t* serializer) const {
2      tail->disambiguate (serializer);
3  }
```

Figure 6.11: Implementation of the `disambiguate` method of the RDQ

Finally, the method returns the node previously at the tail of the list, which serves as the disambiguation node for the new operation.

Figure 6.11 lists the `disambiguate` method, which is called by ordinary delegations to ensure correct ordering with previous ambiguous operations. It takes the serializer named by the delegation as its sole argument, which it uses to invoke the `disambiguate` method of the RDQ node at the tail of the list (line 2).

Figure 6.12 presents the implementation of the `clean` method. Unlike the previously described methods, which are always executed sequentially to prepare for ambiguous delegation, the clean method may be invoked concurrently by multiple worker threads attempting to propagate the *resolved* flag. Therefore this operation requires careful synchronization. The `clean` method updates `resolved` using compare-and-swap (CAS), and only the thread that successfully changes the flag is allowed to continue, excluding other threads from modifying the the RDQ list.

The `resolved` field is implemented with the `tagged_bool_t` type, which operates as a normal boolean value and the field incorporates a tag that is in-

cremented on each modification of the field. The tag is used to avoid the ABA problem that can occur with CAS-based synchronization, first described in the IBM System/370 documentation (IBM, 1975, pg. 310–314). The ABA problem manifests when one thread reads the value of a variable to be A. A second thread also reads the value as A and then uses CAS to set the value to B. If the first thread were to use a CAS to set the variable to some other value, the CAS would fail because the value is no longer A. However, if some intervening operation changes the value back to A, then a CAS by the first thread will succeed. This can cause problems when an algorithm uses CAS and assumes that nothing has changed between the initial read and the CAS, because the value is the same.[2]

The ABA problem can occur in the RDQ when multiple worker threads are executing the cleaning routine. Consider the case where one worker reads the resolved flag of a node for a completed ambiguous operation and sees it set to false. A second worker may also see this value, use CAS to set it to true, and then the node may be popped off the list and recycled. The same node may later be reused for another ambiguous operation, causing its resolved flag to be re-initialized to false. The first worker, operating under the assumption that the node represented the older, now completed operation, could use CAS to set the resolved flag to true, erroneously marking the node as representing a completed computation. The `tagged_bool_t` uses the classic solution described in the IBM System/370 documentation, storing the value in a memory word with an additional tag that is incremented on each update, ensuring that a CAS will fail if there have been intervening updates between the original read and the CAS.

The `clean` method first checks to see if the head node of the RDQ list has any successors (lines 4–5). If it does not, then there is nothing for the method to do and it returns. If there is a successor node, then the method checks to see if the `resolved` flag has been set (line 9. If it has, then some other thread is modifying the list, and the method returns (line 10).

The `clean` method next checks the `complete` method, which indicates if the counter of the node's termination-detecting barrier has reached zero (line 13). If the node is not complete, then this node cannot be removed from the list, and `clean` returns (line 23). If the node is complete, then `clean` attempts to set the node's `resolved` flag to true using a CAS operation (line 15). This update simultaneously increments the tag on the `resolved` field to avoid the ABA problem.

[2] By contrast, the load-linked/store-conditional (LLSC) primitive does not suffer from the ABA problem—any intervening store to the address read by load-linked instruction will cause the store-conditional to fail.

```
1  void rdq_t::clean () {
2      while (true) {
3          // Check for ambiguous nodes
4          rdq_node_t* rdq_node = head->get_next ();
5          if (node == NULL) return;
6
7          // If the head successor is resolved,
8          // another thread is performing clean
9          bool resolved = node->resolved.value ();
10         if (resolved) return;
11
12         // If node is complete, attempt to set resolved flag
13         if (node->complete ()) {
14             // If CAS of resolved flag succeeds, pop old head off the list
15             if (node->resolved.cas (false, true)) {
16                 rdq_node_t* old_head = head;
17                 head = rdq_node;
18                 rdq_node->prev = NULL;
19                 delete (old_head);
20             }
21             else return;
22         }
23         else return;
24     }
25 }
```

Figure 6.12: Implementation of the `clean` method of the RDQ

If the CAS fails, then some other thread successfully performed the CAS operation and will finish the cleaning operation, so the method returns (line 21).

Discussion

The current design of the RDQ is tailored to facilitate efficient parallel loops for data-driven decomposition with a minimal amount of synchronization and contention. As we explore other uses of ambiguous delegation, we anticipate that some aspects of the present design may prove to be inadequate. While this design suffices for parallel loops, it is worth mentioning these potential shortcomings for posterity.

The RDQ facilitates ambiguous delegation of methods to different serializers by stalling later delegations until either the serializer named by a particular delegation has completed the earlier delegations, indicated by its timestamp, or until all

previous delegations complete. This leads to the first potential drawback of the present design— it may cause a delegation to stall even when there are no previous ambiguous delegations that will name this data. For example, consider a parallel loop over a container data structure, followed by a delegation to a serializer that is not held in the container. Because the timestamp of this serializer is never incremented by one of the iteration tasks, the delegation will wait until the loop completes and the resolved flag for its ambiguity node is set. We can envision adding loop indices to each RDQ node to reflect the portion of container being accessed, but then each delegation would have to walk the entire RDQ list, checking to see if it is part of any previous delegations. In the present design, each delegation reads from a single RDQ node (the disambiguation node) and writes to a single RDQ node (the ambiguity node), which greatly simplifies the design.

The other potential shortcoming of the design is its reliance on the `clean` routine to lazily propagate the resolved flag. Ideally, we would like to eagerly propagate the resolved flag as the counter of each node reaches zero. However, this would seem to require that the threads executing the loop tasks synchronize each update to the counter of an RDQ node to precisely determine the thread that completes the last ambiguity, and that thread would then propagate the resolved flag. N atomic instructions would be required for a loop with N iterations, versus the single CAS operation currently used to propagate the resolved flag. In the future, we plan to investigate other mechanisms for detecting completion to overcome this problem.

As we investigate further uses of ambiguous delegation, we plan to revisit the design of the RDQ to identify further opportunities to enhance its functionality and performance. However, we believe the current implementation demonstrates a useful approach to overcoming some of the limitations resulting from the receiver identification problem.

6.4 IMPLEMENTATION OF FOREACH

Having described the implementation of the RDQ, we now show how PROMETHEUS uses it to provide an efficient `foreach` loop. We begin by explaining the modifications needed to incorporate the RDQ into the existing constructs for data-driven decomposition.

The `quiesce` operation, described in Section 4.5, ensures that all method invocations delegated to the serializer of an object complete before an impure method is invoked. This operation must be modified as shown in Figure 6.13 to account for the fact that an empty serializer does not guarantee that all out-

```
1  template <typename C>
2  void quiesce (C& object) {
3      // Get serializer associated with object
4      serializer_t* serializer = object.get_serializer ();
5
6      // Disambiguate serializer using the RDQ
7      rdq_t* rdq = serializer->get_rdq ();
8      if (rdq != NULL) rdq->disambiguate (serializer);
9
10     // Yield processor until serializer is empty
11     while (!serializer->get_tag () != 0) {
12         yield ();
13     }
14 }
```

Figure 6.13: Quiesce function updated to use RDQ

standing method invocations have completed—currently ambiguous delegations may resolve their receiver to be this object and delegate a method invocation to its serializer. We therefore modify the `quiesce` operation to disambiguate the serializer of the object (lines 8). Once it is certain that any ambiguous delegation to the serializer of this object has completed, `quiesce` waits for the serializer to become empty, as in the previous implementation.

Once an ambiguous delegation identifies its receiver, it calls `parallel_delegate` to perform the delegation. This operation requires only a few modifications to the explicit `delegate` operation presented in Section 4.5. It takes one additional argument, the RDQ node that serves as the disambiguation node for this operation. Before inserting the method invocation into the serialization queue of the designated receiver, `parallel_delegate` disambiguates the serializer as shown on line 8. As in explicit delegation, a method invocation object is allocated (line 11) and inserted into the serialization queue via the `produce` method of the serializer (line 14). At this point, the method invocation has been ordered in the serializer, and resolution of this ambiguity is communicated to the RDQ on line 14. The remainder of the `parallel_delegate` operation schedules the serializer for execution in the same way as explicit delegation.

We now have all the necessary pieces to implement the `foreach` loop construct, listed in Figure 6.15. As we described in Section 6.2, this function recursively divides the range of a C++ STL container specified by the `begin` and `end` arguments. The recursion is divided into two pieces: the base case, and the divide-and-conquer

```
1  template <typename C, typename B, typename... Args>
2  void parallel_delegate (rdq_node_t* disambiguation_node,
3                          C& obj,
4                          void (B::* method) (Args...),
5                          Args... args) {
6      // Get serializer associated with obj
7      serializer_t* serializer = obj.get_serializer ();
8      disambiguation_node->disambiguate (serializer);
9
10     // Create an invocation object
11     invocation_t* invocation = new invocation_t (obj, method, args...);
12     // Add invocation to serializer
13     bool schedule = serializer.produce (invocation);
14     disambiguation_node->resolve (serializer);
15
16     // If serializer is not currently scheduled, execute it
17     if (schedule) {
18         invocation_t* invocation = serializer.consume ();
19         do {
20             invocation->execute ();
21             delete invocation;
22             invocation = serializer.consume ();
23         } while (invocation != NULL);
24     }
25 }
```

Figure 6.14: Implementation of the `parallel_delegate` function

case.

The base case (lines 10–15) executes when the recursion reaches a point where the range is less than or equal to the grain size specified by the programmer (line 10). (If the programmer does not specify a grain size, it defaults to one.) It iterates over this range (lines 11–14), using `parallel_delegate` to delegate the desired method to each object in the range (line 13).

The divide-and-conquer case (lines 16–22) first identifies the middle element of the range as a split point (line 17). It creates a task to continue dividing the left side of the range (line 18), and then spawns the task (line 19). The divide-and-conquer case then directly calls `foreach` on the right side of the range. The right side need not be spawned as a task, since it is followed by a call to `sync`, which waits until the task for left side completes.

Because `parallel_delegate` operation performs receiver disambiguation,

```
1  // Parallel loop over a half-open range [begin, end)
2  // Each task is N loop iterations specified by grain_size
3  template <typename T, typename C, typename B>
4  void foreach (size_t grain_size,
5                typename T <C>::iterator begin,
6                typename T <C>::iterator end,
7                void (B::*method) (Args...),
8                Args... args)
9  {
10     if ((end - begin) >= grain_size) {
11         for (typename T <C>::iterator iter = begin;
12              iter != end; ++iter) {
13              parallel_delegate (rdq_node, *iter, method, args);
14         }
15     }
16     else {
17         typename T <C>::iterator split = begin + ((end - begin) / 2);
18         foreach_task_t task (grain_size, begin, split, method, args);
19         spawn (task);
20         foreach (grain_size, split, end, method, args);
21         sync ();
22     }
23 }
```

Figure 6.15: Implementation of the `foreach` loop construct

`foreach` requires no special action to ensure the delegated method invocations are ordered correctly with respect to delegations before and after the loop. The RDQ handles all of the ordering requirements, so that the implementation of `foreach` is essentially identical to a standard divide-and-conquer loop.

6.5 SUMMARY

In this chapter, we described the receiver identification problem, a fundamental limitation of any parallel execution model that dynamically enforces program ordering of operations on each variable or data structure. For some computations, receiver identification may constitute a significant fraction of each operation. Restricting parallel execution to the portion of the computation after receiver identification ensures a correct ordering of such operations, but it may place an unacceptable limit on the amount of parallelism in the program. To overcome this

limitation, we proposed receiver disambiguation, which allows receiver identification to proceed in parallel and preserves program ordering of delegations to a particular receiver. We then described the operation and implementation of the receiver disambiguation queue (RDQ), a novel mechanism for performing receiver disambiguation. To demonstrate the feasibility of this technique, we applied the RDQ to the implementation of divide-and-conquer parallel loops. In the future, we plan to study the application of receiver disambiguation to other forms of computation that are limited by the receiver identification problem.

7 EXPERIMENTAL EVALUATION

> *We're just starting. It's a bit evolutionary and there are an awful lot of chicken-and-egg problems. You have to have parallel computers before you can figure out ways to program them, but you have to have parallel programs before you can build systems that run them well.*
>
> — Chuck Thacker (2010)

In this chapter, we present an experimental evaluation of the PROMETHEUS system for data-driven decomposition. We begin by describing our benchmark applications, and how we applied PROMETHEUS to facilitate a data-driven decomposition to each one (Section 7.1). We then describe our methodology for performing our experiments (Section 7.2) and present the results of this evaluation (Section 7.3), including a comparison of the performance of control- and data-driven decomposition of the same programs. We conclude by summarizing the findings of this analysis (Section 7.4).

7.1 BENCHMARKS

There are many possible questions one might ask when evaluating a new parallel execution model: *Is it easy to learn? Does it have a positive impact on the complexity of developing, debugging, and maintaining software? Are the resulting programs reliable? Can it interact with existing code? Does the model enable parallel execution for new classes of applications?* Unfortunately, these questions do not readily lend themselves to quantification. We rely on the arguments made in Chapter 2 and Chapter 3 to convince the reader that by using a sequential program representation to derive repeatable, predictable parallel execution, data-driven decomposition could potentially answer many of these questions in the affirmative.

Assuming these are desirable goals we seek to answer the following questions with our evaluation: *Can data-driven decomposition achieve performance that is competitive with control-driven decomposition? Can dynamic parallelization provide higher performance for some applications than static parallelization? How effective are the mechanisms for efficient data-driven decomposition proposed in this dissertation?* Answering these questions will help us understand if there are inherent costs associated with the benefits of the model, and allow us to identify areas for future improvement.

Program	Source	Original Language	Description
`barnes-hut`	Lonestar	C++	N-body simulation
`black-scholes`	PARSEC	C++	Financial analysis
`bzip2`	pbzip2	C	Compression
`canneal`	PARSEC	C++	VLSI CAD
`dedup`	PARSEC	C	Enterprise storage
`histogram`	Phoenix	C	Image analysis
`reverse_index`	Phoenix	C	HTML analysis
`word_count`	Phoenix	C	Text processing

Table 7.1: Benchmark Programs

To this end, we perform an evaluation using existing parallel applications to provide a basis of comparison between control- and data-driven decomposition. These applications, which are summarized in Table 7.1, are drawn from the Lonestar (**?**), PARSEC (Bienia et al., 2008), and Phoenix (**?**) benchmark suites. These programs are all written in either C or C++ and parallelized with pthreads. We ported these benchmarks to PROMETHEUS by first rewriting them as idiomatic, object-oriented C++ programs, using standard template library (STL) data structures. We then annotated class specifications and method delegations to facilitate a data-driven decomposition.

When implementing a parallel algorithm in PROMETHEUS, we endeavored to apply data-driven decomposition to the same set of operations as the original program. Some of the benchmark programs do not exploit all opportunities to harness parallelism, and we avoided annotating these operations in our versions. But while we parallelize the same operations, the sequential representation and dynamic decomposition of a PROMETHEUS program may result in a very different parallel execution. For example, a common approach to multithreading a data-parallel algorithm is to read the input data set from a file, and then parcel chunks of this data to a set of threads, which apply a set of operations to each data element in a chunk. By contrast, a PROMETHEUS program can read in the first object and immediately delegate a set of operations to its serializer. As the serializer invokes these methods, the program reads in the next object begins delegating operations to its serializer. Rather than having all data in flight at once, each executing serializer forms a pipeline through which each object flows.

Despite parallelizing the same set of operations, the substantial differences between the two models means that there are sometimes different amounts of parallelism in the two versions of each application. There are two primary sources of these differences. First, because we apply data-driven decomposition at the level of objects, it is naturally more fine-grained than many multithreaded programs. This helps some programs by exposing additional parallelism, but it harms others because parallelization overheads must be amortized over shorter computations. Second, there are some cases where multithreaded programs employ shared data and synchronization in ways that have no natural analogue in a data-driven decomposition. In these cases, we had no choice but to take a different approach. We will highlight any important differences in our discussion of the results.

Having explained our overall approach to applying data-driven decomposition to existing parallel applications, we now turn to the individual benchmarks. We will give a brief description of each, and simplified code showing how they are expressed using PROMETHEUS.

Barnes-Hut

Description. The `barnes-hut` benchmark is an N-body simulation, which calculates the motion of a group of N particles that interact with each other via forces such as gravity (**?**). The Barnes-Hut algorithm avoids computing all $O(N^2)$ interactions between particles using a spatial partitioning structure called an octree. Each node in the octree represents a cell in three-dimensional space, and summarizes the mass and center of gravity of all particles it contains. Using the octree, the program need only compute direct interactions of nearby particles, and uses the cell summaries for more distant particles, reducing the complexity of the algorithm to $O(N \log N)$. The Barnes-Hut algorithm iterates over time steps, performing the following actions for each step: (1) create a new octree and insert all the bodies, (2) compute the force acting upon each body by traversing the octree, and (3) update the position and velocity of each body based on the incident force.

Pthreads implementation. We evaluate the Lonestar (**?**) implementation of `barnes-hut`. While all three of the steps in the program are amenable to parallelization, the Lonestar version parallelizes only the second step—computing the force incident on each object using the octree—which dominates the execution time of the program (**?**). The program creates a set of threads and assigns an equal

```
1  typedef prometheus::private_t <force_t> private_force_t;
2
3  void barnes_hut (vector <body_t>& bodies, int n_timesteps) {
4      vector <force_t> forces (bodies.size ());
5
6      prometheus::begin_delegation ();
7      for (int timestep=0; timestep < n_timesteps; ++timestep) {
8          octree_t octree;
9          for (size_t i = 0; i < bodies.size (); ++i) {
10             octree.insert (bodies[i]);
11         }
12         octree.summarize_subtrees ();
13
14        private_force_t::foreach (forces.begin (), force.end (),
15                                  &force_t::compute_force, octree);
16
17        for (size_t i = 0; i < bodies.sizes (); ++i) {
18             bodies[i].advance (forces);
19         }
20     }
21     prometheus::end_delegation ();
22 }
```

Figure 7.1: PROMETHEUS pseudo-code for `barnes-hut`

number of bodies to each thread. The threads then compute the forces on each body they are assigned. Barrier synchronization ensures that the third step of the program does not begin updating the position and velocity of the bodies until the second step has completed computing the incident forces.

PROMETHEUS implementation. Conventional implementations of Barnes-Hut store all information about a body, including its mass, position, velocity, and incident force in a single data structure or object. This organization of data is not amenable to parallelization with PROMETHEUS because in the second phase of each timestamp, the operation computing the forces on that object writes to the force fields of the object, while other operations may read the position and mass of that object. Because the values that are written are disjoint from the values that are read, this does not introduce a determinacy race. However, because PROMETHEUS requires objects to be operated on by one operation at a time, we split this object into a body object holding the mass, position, and velocity of the

body, and a corresponding force object that stores the force information. Thus during the second phase of each timestep, the force objects are accessed by a single operation, while the bodies, which are constant during the phase, are read by multiple operations.

Figure 7.1 lists pseudo-code for the PROMETHEUS implementation of`barnes-hut`. The program uses two primary classes of objects: a `body_t` object stores the mass and velocity of a particle, while a `force_t` object is used to compute the force acting upon a particular body. We indicate that the `force_t` class will be used to create private objects using the private wrapper on line 1. Inside the `barnes_hut` function we create a delegation region (lines 6–21). We then use a loop to iterate over the specified range of time steps (lines 7–20).

The first step of the algorithm builds the octree by inserting all bodies into the octree (lines 8–11). The last part of this step summarizes the contents of each cell in the octree (line 12). The second step of the algorithm computes the force acting on each body. In our program, we use the `foreach` loop to delegate `compute_force` to each `force_t` object (lines 14–15). In this phase of the computation, the octree is read-only, and only the `force_t` objects are being written. The third step of the algorithm modifies the velocity and updates the position of each body object based on the corresponding force object (lines 17–lines 19). Note that when the code in the `advance` method (line 18) executes, it uses the `call` interface of the `private_force_t` class, which will implicitly synchronize the serializer of the force object accessing it.

Black-Scholes

Description. An *option* is a financial instrument that provide the owner with the right to buy or sell an asset at a set price. The Black-Scholes partial differential equation (PDE) (**?**) is commonly used to determine the value of European-style options.[1] The `black-scholes` application uses this PDE to compute the prices for portfolios of European options.

Pthreads implementation. The `black-scholes` application provided in the PARSEC benchmark suite uses the Black-Scholes PDE to compute the prices of a portfolio of options (Bienia et al., 2008). The price computations of different options are independent, and thus each option in the portfolio presents an oppor-

[1] European-style options may be exercised only on a given date, whereas American-style options may be exercised any time on or before a given date.

```
1  typedef prometheus::private_t <option_t> private_option_t;
2  typedef prometheus::private_t <output_file_t> private_output_file_t;
3
4  void blackscholes (input_file_t& input_file, int num_options) {
5      private_output_file_t output_file (output_filename);
6      private_option_t* options = new private_option_t[num_options];
7
8      prometheus::begin_delegation ();
9      for (int i = 0; i < num_options; ++i) {
10         options[i].call (&option_t::initialize, input_file);
11         options[i].delegate (&option_t::compute_price);
12         options[i].delegate (&option_t::output, output_file);
13     }
14     prometheus::end_delegation ();
15 }
```

Figure 7.2: PROMETHEUS pseudo-code for `black-scholes`

tunity for parallel execution. The multithreaded version of `black-scholes` reads in a number of options from a file, and divides these options into equal-sized sets, each of which is assigned to a thread for execution. Once it determines the price of all the options, indicated via barrier synchronization, the program writes the results out to a file.

PROMETHEUS implementation. Like the pthreads implementation, our PROMETHEUS version also parallelizes the pricing of different options. But rather than grouping options together for parallel execution, the PROMETHEUS version parallelizes the operations on individual options. This affords it three main advantages over the pthreads version: First, reading the input file and writing the output file contribute a significant amount of sequential processing time to the application. The PROMETHEUS implementation is able to mitigate this bottleneck by immediately beginning parallel execution of the pricing of each option as it is read in, and writing out the results of each operation once it and all previous operations have completed. Second, reading all of the options before pricing them coupled with pricing all of the options before writing out the results destroys the temporal locality of operations on an individual option. Using a finer-grained parallelization afforded by PROMETHEUS, `black-scholes` can foster locality by performing operations on each option more closely together in time. Third, parallelizing the pricing of individual operations allows the dynamic scheduler to

```
1  typedef prometheus::private_t <block_t> private_block_t;
2  typedef prometheus::private_t <output_file_t> private_output_file_t;
3  void compress (input_file_t& input_file, char* outfile) {
4      private_output_file_t output_file (outfile);
5
6      prometheus::begin_delegation ();
7      while (!input_file.empty ()) {
8          private_block_t* block = new block (input_file);
9          block.delegate (&block_t::compress);
10         block.write (output_file);
11     }
12     prometheus::end_delegation ();
13 }
```

Figure 7.3: PROMETHEUS pseudo-code for `bzip2`

balance the load among worker threads.

FIgure 7.2 sketches a PROMETHEUS implementation of `blackscholes`. For this program, we use two classes of private objects: the `private_option_t` class defined on line 1, and the `private_output_file_t` class defined on line 2. After initializing the output file (line 5) and an array to hold each option (line 6), we declare a delegation region (lines 8–14). We employ a loop to iterate over each option (line 9–13). Inside the loop, we initialize the option by reading in data from the input file (line 10). We then delegate the `compute_price` method on line 11. Finally, we delegate the `output` method, which takes the output file as an argument (line 12). Because the output file is a private object, PROMETHEUS automatically performs multiple delegation (as described in Section 3.4). This allows the program to continue without waiting for output to the file to complete, while ensuring that each output method accesses the file in program order.

Bzip2

Description. The popular `bzip2` compression utility implements the Burrows-Wheeler Transform (BWT) (**?**), a block-sorting, lossless data compression algorithm. The BWT takes blocks of a predetermined size, and applies a reversible transformation that greatly improves the compressibility of the block. Then `bzip2` uses a secondary algorithm, such as Huffman Encoding, to compress the block.

Pthreads implementation. `Pbzip2` is a popular parallel implementation of `bzip2` developed by **?**. `Pbzip2` uses a pipeline-style parallelization to compress independent blocks from the file. The first step in the pipeline is reading in the blocks from a file. This step is performed by the original program thread, which reads each block of the input, and records the block number and a pointer to the block in a shared global array protected by a mutex lock. The original thread then inserts the block pointer into a shared producer-consumer queue. Queue accesses are synchronized with a mutex lock, and condition variables are used to indicate the full and empty states.

The second step in the pipeline is implemented using a set of threads, which each repeatedly retrieve a block from the queue and compress it. The compression thread then inserts a pointer to the compressed data into the global table under the protection its mutex lock.

The third step in the pipeline is writing the blocks out to a file, which is performed by a single dedicated thread. This thread decouples the writing of the output file from the reading of the input file so they can be overlapped in time. The output thread identifies the next block in the input ordering by monitoring its entry in the shared global table, which it accesses under the protection of its mutex lock. Once the compressed data arrives the output thread proceeds to monitor the next entry in the table.

PROMETHEUS implementation. Compression utilities are frequently used to compress large files. The advantage of the pipelined implementation of `pbzip2` is that it greatly reduces the memory footprint of parallel compression by only having a small number of blocks in flight at once, rather than trying to compress the entire file in parallel. We achieve similar benefits using PROMETHEUS by reading each block sequentially and delegating a method to compress that block to its serializer. We also decouple writing the output by creating an object for the output file, and delegating a method to write each block to the file. Multiple delegation (Section 3.4) automatically enqueues this method invocation in the serializers of both the block and the output file, preserving the order that blocks are written out. By contrast with the pthreads implementation, this pipeline does not require any extra work on the part of the programmer to implement and synchronize queues and bookkeeping tables. Instead, it is implicit in the data-driven decomposition, and is automatically implemented by the serializers associated with the block and output file objects.

Figure 7.3 lists pseudo-code for the PROMETHEUS implementation of `bzip2`.

This program uses two classes of private objects: `block_t` objects used to store and compress data (line 1), and a single `output_file_t` object to write out the compressed data to a file.

After opening the output file (line 4) and declaring a delegation region (lines 6–12), the program reads block from the input file in a loop until the file is empty (lines 7–11). Inside the loop, the program reads a block from the input file (line 8), and delegates the `compress` method (line 9). Finally, it uses multiple delegation to perform the block write (line 10), which ensures the method is ordered correctly on both the objects for both the block and the output file.

Canneal

Description. Simulated annealing is a technique for solving the global optimization problem in a large search space. The `canneal` benchmark simulates the application of simulated annealing to finding the optimal routing cost for a chip design. This program is part of the PARSEC benchmark suite (Bienia et al., 2008).

The `canneal` randomly chooses a pair of elements from a supplied netlist and evaluates the effect of swapping them on the routing cost of the chip. If swapping the elements reduces the cost, then elements are swapped. With the simulated annealing algorithm, elements that result in an increased cost are accepted according to a probability. This allows the optimization to escape local minima in order to reach the global minimum routing cost.

Pthreads Implementation. The pthreads implementation of `canneal` spawns multiple threads to simultaneously evaluate the effects of swapping elements in the netlist. Swaps are performed using atomic instructions to ensure elements are not lost from the netlist, but this synchronization strategy does not prevent concurrent swaps involving a common element from resulting in an unintended swap. Instead, such unintended swaps are assumed to be relatively rare, and thus the program relies on the optimization of simulated annealing to recover from the change.

PROMETHEUS Implementation. The PROMETHEUS implementation of `canneal` divides the netlist into a number of regions, which are represented as private objects (line 1). Pointers to these regions are stored in an array (line 4). For each step in the simulation, the program delegates the `swap` method using multiple delegation on each pair of regions in the array (lines 11–13). Note that this method

```
1  typedef prometheus::private_t <region_t> private_region_t;
2
3  void canneal (temperature_t temperature, netlist_t netlist) {
4      vector <region_t> regions = netlist.get_regions ();
5      while (!test_convergence ()) {
6          temperature = temperature / 1.5;
7          while (completed_moves < moves_per_temp) {
8
9              prometheus::begin_delegation ();
10             for (int i = 0; i < regions.size (); i += 2) {
11                 private_region_t* region1 = regions[i];
12                 private_region_t* region2 = regions[i+1];
13                 region1->delegate (&region_t::swap, *region2);
14             }
15             prometheus::end_delegation ();
16
17             for (int i = 0; i < regions.size (); i++) {
18                 regions[i]->call (&region_t::update_locations);
19             }
20             next_permutation (regions.begin (), regions.end ());
21         }
22     }
23 }
```

Figure 7.4: PROMETHEUS pseudo-code for `canneal`

is delegated to the serializer of both regions, and only considers swapping netlist elements with the two regions. During each step of the simulation, the list of regions is permuted, so that the next swap consideres different pairs of regions (line 20).

Each region object is solely responsible for performing updates to its portion of the netlist. However, swaps of netlist elements may also affect the routing cost of the fan-in and fan-out nodes of a netlist element, which may lie outside the region. Therefore swaps are not performed immediately, but instead recorded locally in each region object. Once each pair of regions evaluates its potential swaps, the actual swaps are performed sequentially to avoid creating determinacy races (lines 17–19).

Dedup

Description. Data duplication is a significant contributor to the overall data footprint of enterprise storage systems, due to redundant copies of files, email, and other data. Deduplication is an emerging technique for compressing backup storage systems by eliminating these redundant copies of identical blocks of data. Used in conjunction with other mechanisms for data reduction, such as delta encoding (i.e., "diffs"), and conventional compression, deduplication can greatly reduce the cost of data storage, backup and archival.

Deduplication relies on collision-resistant hash functions with a sufficiently large output range to generate a fingerprint for a block of data(**?**). A popular choice of hash function for deduplication is the SHA1 cryptographic hash function (**?**), which uses 160 bit keys and yields a probability of 10^{20} in an exabyte (10^{18} bytes) of data. This probability is so low that an (**?**) may be safely treated as a unique identifier for a the data. Thus duplicate copies of a storage block can be replaced with the hash key, so that only one copy of the block is required.

Pthreads implementation. The `dedup` benchmark is a parallel implementation of deduplication included in the PARSEC. The program breaks an input file into a set of coarse chunks to create independent units of work, and then divides each of these coarse chunks into a set of fine-grained fragments using fingerprinting (**?**) to reduce the probability of fragmenting duplicate sequences in the data. Next, `dedup` computes the SHA1 hash of each fragment, and checks a global hash table to see if a fragment with the same hash code has been encountered, indicating the current fragment is a duplicate. If the fragment is a duplicate, only its fingerprint is written to the output file. If the fragment is not a duplicate, it is added to hash table, compressed, and the compressed version is written to the file.

Like the previous `pbzip2` example, `dedup` uses a pipeline approach to parallelization. The first stage of the pipeline, implemented in a single thread, reads the input file and create the coarse chunks from its data. The second stage uses multiple threads to parallelize the fingerprinting and subdivision of the coarse-grained chunks into fragments. The third stage also uses multiple threads, parallelizing the compression of the fragments generated in the second stage. Finally, the last stage uses a single thread to reconstruct the original order of the fragments and write them to the output file.

Different pipeline stages communicate uses FIFO producer-consumer queues, synchronized with mutex locks and condition variables. Accesses to the global

hash table, which is shared among the threads constituting the third and fourth stages of the pipeline, are synchronized using mutex locks. These accesses can occur in different orders when the program is run multiple times on the same file, due to the particular interleaving these threads. Therefore the first time an instance of a duplicate fragment is encountered may not be the first instance of that duplicate in the input file. Consequently, the compressed instance of the duplicate is not always the first instance in the output file—it may be preceded by one or more fingerprints, meaning that the output of PARSEC `dedup` is non-deterministic.

PROMETHEUS Implementation. Our PROMETHEUS implementation of deduplication uses private objects for the coarse-grained chunks as well as the fine-grained fragments. Initially, the program reads in data from the input file and creates a private object for each chunk. Next, it delegates a method to fingerprint and fragment each chunk. It then sequentially accesses the hash table, which maps fingerprints to compressed data, to determine which fragments need to be compressed. The PROMETHEUS version of `dedup` then delegates a method to compress each uncompressed fragment. Finally, it writes each fragment to an output file.

As with `bzip2`, the PROMETHEUS version of `dedup` results in a pipeline of data flowing through the program, without requiring the programmer to implement the low-level details of its execution. Furthermore, because it ensures a sequential ordering of accesses to the hash table, it always produces the same output file: the first instance of a duplicate fragment is always the compressed copy, and subsequent duplicates are always fingerprints. Not only does this make the program repeatable and easier to debug, it also simplifies decoding the compressed file, because a fingerprint is never encountered before the corresponding data is known.

Figure 7.5 lists simplified code for the PROMETHEUS version of `dedup`. It employs two classes of private objects: `private_chunk_t` for coarse chunk objects (line 1) and `private_fragment_t` for fragments (line 2). The program tracks which fragments have been previously compressed using the hash table on line 5.

The program reads from an input file in a loop (lines 6–22). In the first phase of the program, the program creates a set of data chunks by reading them in from an input file (line 10). As each chunk is read, the program delegates the `find_fragments` method, which uses the Rabin-Karp fingerprinting technique to identify good splitting points in the data. Once all chunks are complete, the program proceeds to its second phase.

In the second phase of the program, the loop on lines 15–17 calls the `compress_fragments` method of the `chunk_t` class, listed on lines 25–34. This

```
1  typedef prometheus::private_t <chunk_t> private_chunk_t;
2  typedef prometheus::private_t <fragment_t>private_fragment_t;
3
4  void dedup (input_file_t& input_file) {
5      hash_table <private_fragment_t*> compressed_frags;
6      do {
7          prometheus::begin_delegation ();
8          private_chunk_t* chunks[CHUNK_SET_SIZE];
9          for (int i = 0; i < chunks.size (); ++i) {
10            chunks[i] = new chunk (input_file);
11            chunk->delegate (&chunk_t::find_fragments);
12         }
13         prometheus::end_delegation ();
14
15         for (int i = 0; i < chunks.size (); ++i) {
16             chunks[i]->compress_fragments (compressed);
17         }
18
19         for (int i = 0; i < chunks.size (); ++i) {
20             chunk[i]->write (output_file);
21         }
22     } while (!input_file.eof ());
23 }
24
25 chunk_t::compress_fragments (hash_table <fragment_t*> compressed) {
26     prometheus::begin_delegation ();
27     for (int i = 0; i < num_fragments; ++i) {
28         if (!compressed.find (fragments[i]) {
29             compressed->insert (fragment[i]);
30             fragment[i]->delegate (&fragment_t::compress);
31        }
32     }
33     prometheus::end_delegation ();
34 }
```

Figure 7.5: PROMETHEUS pseudo-code for dedup

method loops over each of the fragments it identified in the previous phase, and any fragment that is not found in the hash table of compressed fragments is inserted into the hash table, and the `compress` method is delegated to the serializer of that object (line 30).

When the method returns, the program enters its third phase, which writes the compressed copy or fingerprint for each fragment (lines 19–21).

Histogram

Description. The `histogram` benchmark is a simple image processing program that analyzes a bitmap image to determine the frequency each value for the red, green, and blue components of each pixel. We adapted this program from a suite of benchmarks originally developed for the Phoenix multi-core MapReduce system (**?**).

Pthreads implementation. The pthreads version of `histogram` reads a bitmap image file into memory. It then spawns a set of threads and assigns part of the image file to that thread. A thread examines all of the pixels in its portion of the file, incrementing one of 256 counters for the 8-bit values of each of the red, blue, and green components of the pixel. Once all of the threads have completed, the resulting histograms are accumulated sequentially to produce a single histogram.

PROMETHEUS Implementation. The PROMETHEUS implementation of `histogram` divides the bitmap file into chunks, and associates each chunk with a private object that maintains the histogram for that chunk. It delegates a the method to compute the histogram for each chunk, and once these methods complete, the resulting histogram objects are combined to produce the final result.

Of the applications we studied for this dissertation, `histogram` resulted in the strongest similarity between the pthreads and PROMETHEUS implementations. Due to the simplicity of the algorithm, the control-driven and data-driven decompositions are essentially the same—the delegated method invocations correspond exactly to the actions of threads. However, the PROMETHEUS version still has two advantages: First, the decomposition is expressed using annotations on an sequential program, without introducing the cruft of configuring threads and marshaling their arguments. Second, the ability to easily select the size of the bitmap chunk associated with each histogram affords the programmer the ability to choose a parallelization granularity that is high enough to amortize the parallelization over-

```
1  typedef prometheus::private_t <histo_t> private_histo_t;
2
3  void histogram (input_file_t& input_file) {
4      bmp_t bmp (input_file);
5      private_histo_t* histos = new histo_t[NUM_CHUNKS];
6      prometheus::begin_delegation ();
7
8      int num_chunks = bmp.size () / CHUNK_SIZE;
9      int start = 0;
10     for (int i = 0; i < num_chunks; ++i) {
11         histos[i] = new private_histo_t (start, start + chunk_size);
12         histos[i].delegate (&histo_t::calculate);
13         start = start + CHUNK_SIZE;
14     }
15     prometheus::end_delegation ();
16
17     summarize (histos)
18 }
```

Figure 7.6: PROMETHEUS pseudo-code for `histogram`

heads, but small enough to allow the runtime the flexibility to perform dynamic load balancing.

Figure 7.6 lists the PROMETHEUS code for `histogram`. It uses one class of private object, the `private_histo_t` class, defined on line 1. When the program begins, it first initializes a bitmap image by reading it from a file (line 4), and initializes an array of histogram objects (line 5). The program divides the bitmap into a number of equally sized chunks, and creates a histogram object for each chunk (line 11). It then delegates the `calculate` method to each of these objects. Upon completion of these methods, the program sequentially summarizes the results.

Reverse Index

Description. A *linked database* comprises a set of nodes that are linked together in some way, such as web pages via hyperlinks, or academic papers via citations. Counting the number of links to a particular node in a linked database provides a metric that is useful in determining that node's importance. For example, the Google PageRank algorithm uses the number of links to a particular web page as

one metric for determining its relevance to a web search (**?**). The metric weights each link based on the estimated importance of its source.

To compute the rank of a web page, it is therefore necessary to know how many web pages link to it, as well as address of the linking web page. The `reverse_index` application from the Phoenix suite (**?**) performs this function by analyzing a set of web pages and extracting the links within each page. It uses this information to create an index that indicates for each web page, the set of web pages that link to it.

Pthreads implementation. The pthreads version of `reverse_index` provided in the Phoenix suite processes web pages stored on disk by a web crawler. It recursively traverses the directory tree, and each time it encounters an HTML file, it adds that file to a linked list. Once the files have all been identified, it spawns a set of threads to find the links in each web page. A thread begins this process by dequeing the file currently at the front of the list, using a mutex lock to preserve the consistency of the list. This approach differs from the other multithreaded applications we have examined, because each thread dynamically acquires work as it runs, rather than being assigned a predetermined set of data. This dynamic distribution of work balances the load on each thread.

Once a thread has acquired the source file for a web page, it scans it to identify links to other pages. The thread tracks information about these links using a thread-local list which contains one entry for the address it has encountered a link to, and the list of files that contained that link. The list is kept sorted according to the lexicographic ordering of the link address for each entry. This sorting adds extra overhead when a new link is inserted, but simplifies merging the results of each thread.

Once all the files have been processed, Phoenix `reverse_index` merges the results of the threads. It spawns a thread for each pair of lists that merges their contents together. The fact that the lists are sorted simplifies the merging process because the lists can both be traversed in order, rather than traversing one list and searching for matches in the second list. The merging process is repeated recursively, dividing the number of threads by two at each step until there is a single, final list containing the index mapping each web address to the set of addresses linking to it.

PROMETHEUS Reductions. Both `reverse_index` and `word_count` perform independent computations on private objects, and then need to summarize the results of these operations to produce their final result. Since these programs may

both involve potentially very large data sets, performing these summarizations sequentially could introduce a significant sequential bottleneck that would limit the performance of the program. To overcome this limitation, we implemented a set of container data structures that support *reductions* (sometimes referred to as *folds*, which uses an associative operator to combine data elements to produce a final value. We then use PROMETHEUS to perform parallel reductions over these containers to speed up the summarization process.

There are two key components of these parallel reductions. The first is the concept of a *reducible object*, which is an object that specifies a way for it to be combined with another object of the same class. To make an object reducible, programmers need only specify a `reduce` method that takes another object of the same class as an argument, and performs the desired summarization of the data. The second component of our parallel reductions are *reducible containers* for holding reducible objects. These containers maintain local copies of a container for each worker thread in the system, which allows delegated method invocations to operate on the container without introducing determinacy races. The containers also provide a `reduce` method to combine the local copies of the container into a single container holding the result. When duplicate reducible objects are detected in different local copies, they are combined using the `reduce` method specified by the programmer.

We parallelized the container reductions using PROMETHEUS. The parallel reduction recursively combines pairs of local containers until there is only one, final container of reducible objects. For example, if a program running with eight worker threads were to reduce the container, the first step combines local containers 0,1, 2,3, 4,5 and 6,7 in parallel; the second step combines local containers 0,2 and 4,6 in parallel; and the third step would combine local containers 0,4. When the reduction completes, container 0 holds the summary result.

PROMETHEUS Implementation. The PROMETHEUS version of `reverse_index` uses two key classes of objects. Files are represented using private objects, and a `find_links` method is delegated on each file. Links are represented using reducible objects. Each link object stores a set containing the files in which it has been encountered. The `reduce` method of the link object takes another link as an argument, and adds all the files in that link's file set to its own.

PROMETHEUS `reverse_index` also traverses the directory structure specified by the input, finding HTML files. However, rather than waiting until it has found all files, the PROMETHEUS implementation immediately creates a private object for

each file and delegates `find_links`. When this method encounters a link in the file, it uses the address string to get the link object associated with that address from a reducible version of the STL `map` container. Once all of the delegated invocations of `find_links` completes, the reducible `map` is reduced to produce a single `map` container holding the results of the program: a map from a text address of a web page to a set of files that contain that link.

The PROMETHEUS implementation of `reverse_index`, listed in Figure 7.7, uses a single class of private object, the `private_file_t` class defined on line 1. For convenience, the program also declares a set container for files (line 2) which is instantiated on line 6, and declares a container for mapping from link text to link objects, which is instantiated on line 7.

The program begins by calling the `find_files` function on line 10, which is listed on lines 17–25. This function recurses the directory tree, allocating a new file object and delegating the `find_links` method each time it encounters a file (line 19–line 20).

The `find_links` method (lines 27–37) scans through the file to find each link (lines 28–29). If the link has been encountered before, the file is added to the set of files associated with that link (lines 30–31). If the link has not been encountered before, a new one is created, the file is added to its file set, and then the link is added to the link map (lines 33–34).

Once `find_files` has located all the files, it returns. The program then performs a reduction on the `link_map` to summarize the results (line 13). Finally, the link map is written out to a file (line 14).

Word Count

Description The `word_count` application is a utility that counts the number of times each particular word appears in a file, and prints out the 10 most frequently occurring words. This benchmark is part of the Phoenix suite (**?**).

Pthreads Implementation The pthreads implementation of `word_count` reads the specified file into memory, and spawns a set of threads. The file is divided into equally sized chunks, taking to only split it between words, and each chunk is assigned to a thread. Each thread scans through its chunk of the file, identifying individual words in the text. The threads track each word and its associated count using a thread-local sorted list, similar to the sorted list of files used by Phoenix `reverse_index`. After the file is processed, another set of threads is spawned to

```
1  typedef prometheus::private_t <file_t> private_file_t;
2  typedef set <private_file_t*> file_set_t;
3  typedef reducible_map <string, link_t*> link_map_t;
4
5  int reverse_index (const char* path, output_file_t& output_file) {
6      file_set_t file_set;
7      link_map_t link_map;
8
9      prometheus::begin_delegation ():
10     find_files (root_dir, file_set, link_map);
11     prometheus::end_delegation ();
12
13     link_map.reduce ():
14     output_file.write (link_map);
15 }
16
17 void find_files (const char* path, link_map_t& link_map) {
18     if (is_file (path)) {
19         private_file_t* file = new private_file_t (path);
20         file->delegate (&file_t::find_links, link_map);
21     }
22     else { // path is a directory
23         // open directory and recurse on contents
24     }
25 }
26
27 file_t::find_links (link_map_t& link_map) {
28     while (!eof ()) {
29         const char* link_text = find_next_link ();
30         if (link_map.find_text (link_text)) {
31             link_map[link_text]->add_file (this);
32         else {
33             link_t* link = new link_t (link_text, file);
34             link_map.insert (link_text, link);
35         }
36     }
37 }
```

Figure 7.7: PROMETHEUS pseudo-code for `reverse_index`

```
1  typedef prometheus::private_t <string_buffer_t> private_buffer_t;
2  typedef reducible_map <char*, count_t> count_map_t;
3  typedef vector <private_buffer_t*> buffer_vector_t;
4
5  void word_count (input_file_t input_file, int num_results) {
6      buffer_vector_t buffers;
7      count_map_t count_map;
8      prometheus::begin_delegation ();
9      while (!input_file.eof ()) {
10         private_buffer_t* private_buffer =
11             new private_buffer_t (BUFFER_SIZE);
12         private_buffer->fill_on_word_boundary (input_file);
13         private_buffer->delegate (&string_buffer_t::find_words,
14                                   count_map);
15     }
16     prometheus::end_delegation ();
17
18     count_map.reduce ();
19     print_top_n (count_map, num_results);
20 }
```

Figure 7.8: PROMETHEUS pseudo-code for `word_count`

merge the sorted lists of words, accumulating their counts to produce a single list. This list is sorted by count in decreasing order, and the first ten words and their counts are printed to the screen.

PROMETHEUS Implementation The PROMETHEUS implementation of `word_count` uses string buffer objects to hold pieces of the input file, and a counter object to count the number to track the number of occurrences of each individual word. The counters are stored in a reducible map that that associates a particular text string with the associated counter for that word.

PROMETHEUS `word_count` reads in chunks of the input file of the specified size, and creates a new string buffer object for each chunk. It then delegates the `find_words` method, which scans through the string buffer, incrementing the counter for each word it finds. Once the entire input file has been processed, the map holding the counters is reduced to summarize the results of the parallel computations.

Figure 7.8 lists the pseudo-code for the PROMETHEUS implementation of `word_count`. The program uses private objects for the `string_buffer_t` class

Program	Original Parallelization	Original Synchronization	PROMETHEUS Parallelization
`barnes-hut`	data-parallel	barrier	data-parallel
`black-scholes`	data-parallel	barrier	pipeline
`bzip2`	pipeline	mutexes, cond. variables	pipeline
`canneal`	data-parallel	atomics	data-parallel
`dedup`	pipeline	mutexes, cond. variables	pipeline
`histogram`	data-parallel	barrier	data-parallel
`reverse_index`	data-parallel	mutexes	pipeline
`word_count`	data-parallel	barrier	data-parallel

Table 7.2: Benchmark Characteristics

(line 1), as well as a reducible map to associate the text string for a word with its associated counter object (line 7).

The program reads the the input file in a loop until it has read the entire file (lines 9–lines 15). Each loop iteration creates a new instance of `private_buffer_t` (line 11), and fills the buffer from the input file using the `fill_on_word_boundary` function, which takes care not to split words. Then the program delegates the `find_words` method to the string buffer object. After the entire file has been read and the delegated method invocations have completed, the counter map is reduced to summarize the results (line 18), and then passed to a function that prints out the specified number of words (line 19).

Benchmark Summary

Table 7.2 lists the benchmarks we use in our evaluation and lists how each program was parallelized, as well as the synchronization used in each program. We also list the approach we used for our PROMETHEUS version of each benchmark. For the data-parallel programs, the corresponding PROMETHEUS program is also data-parallel, using loops to parallelize operations in different data. For the programs that use mutual exclusion, we found that a natural way to express the parallelism in PROMETHEUS was to use a pipeline idiom. As an example, consider a program where each operation manipulates private data and then updates a shared resource. In PROMETHEUS, such a program would be expressed by delegating the indepen-

	AMD Barcelona			Intel Nehalem	
Processor	Phenom 9850	Opteron 8350	Opteron 8356	Core i7 965	Xeon X5550
Sockets	1	4	8	1	2
Cores	4	4	4	4	4
Threads	1	1	1	2	2
Total Contexts	4	16	32	8	16
Clock (GHz)	2.5	2.0	2.3	3.2	2.66
Memory (GB)	8	16	80	12	24
Linux kernel	2.6.18	2.6.25	2.6.25	2.6.18	2.6.18

Table 7.3: Hardware Configurations

dent operations to the serializers of objects containing the private data, and then performing the updates on the shared resource sequentially as the operations complete. These pipelines can also use multiple delegation, instead of sequential execution, if the shared resource is sufficiently encapsulated.

7.2 EXPERIMENTAL METHODOLOGY

Hardware

We performed our evaluation on five different 64-bit x86 processors, including both multi-core and multi-socket systems based on the AMD Barcelona and the Intel Nehalem architectures. For the Barcelona architecture, which integrates four cores per processor, we used a Phenom multi-core system, an Opteron four-socket server with a total of 16 cores, and an Opteron eight-socket server with a total of 32 cores. In the following results, these systems are labeled Barcelona (4), Barcelona (16), and Barcelona (32), respectively. For the Nehalem architecture, which provides four cores per processor, each supporting two thread contexts, we used a Core i7 multi-core system and a Xeon two-socket server. These systems are labeled Nehalem (4x2) and Nehalem (8x2) in our results. Table 7.3 lists the details of the hardware configuration for each of these systems.

	Input Data			
Program	Small	Medium	Large	Units
`barnes-hut`	1,000 / 25	10,000 / 50	100,000 / 75	bodies / steps
`black-scholes`	16,384	65,536	10,000,000	options
`bzip2`	31	185	673	file (MB)
`canneal`	200,000	400,000	2,500,000	netlist elements
`dedup`	31	185	673	file (MB)
`histogram`	100	400	1,4000	bitmap (MB)
`reverse_index`	100	500	1,000	directory (MB)
`word_count`	10	50	100	file (MB)

Table 7.4: Benchmark Inputs

Software

We performed our experiments using the benchmarks described in Section 7.1. We compiled both the pthread and PROMETHEUS versions with GCC 4.4.3 using the highest optimization level `-O3`, and the appropriate tuning flag for each architecture: `-march=amdfam10` for the Barcelona processors, and `-march=core2` for the Nehalem processor. We compiled all programs to 64-bit binaries so that they could utilize the full amount of system memory and exploit the expanded x86-64 register set.

The PROMETHEUS programs used the system default for each configuration parameter. PROMETHEUS automatically detects the number of hardware contexts in a system and uses it as the default number of worker threads. It determines the size of each stacklet based on the default size for pthread stacks, which are 8 kB in Linux. Each work-stealing deque is initially contains 32 entries, where each entry is a pointer to a continuation object (8 bytes).

For our performance results, we measured the execution time of the full program, rather than just the parallel regions. We did this for two reasons: first, it provides a more realistic picture of real-world performance; second, control- and data-driven decomposition can result in different parallel regions, and thus the results would not be comparable. We ran each program ten times, and used the minimum execution time from those runs.

7.3 EVALUATION

In this section, we compare the performance of control-driven and data-driven decomposition by comparing applications implemented in both pthreads and PROMETHEUS. We also characterize several aspects of the PROMETHEUS runtime. We begin by examining the trade-off between performance and design complexity of PROMETHEUS introduced by dynamic memory allocation.

Dynamic Memory Allocation Micro-benchmark

The implementations of serializers and the PROMETHEUS runtime described in the previous chapters use many different dynamically allocated data structures during the execution of the program, including method invocations, serializer list nodes, contexts, and stacklets. The objects are often allocated and deallocated in rapid succession, which would place a heavy burden on the memory allocator. Therefore the PROMETHEUS runtime manages allocation of these data structures using thread-local free lists, circumventing the memory allocator. This strategy prevents serializers and PROMETHEUS from being hampered by the scalability of the memory allocator, but it complicates the implementation of these structures.

To justify this design decision, we developed a micro-benchmark that simulates an extreme version of the allocation behavior of PROMETHEUS. The micro-benchmark spawns a thread for each processor in the system, and then each thread proceeds to repeatedly allocate and deallocate N 64-byte objects in a loop. We measured the performance of the thread-local free lists used by the PROMETHEUS runtime against the standard `glibc` memory allocator that ships with Linux, as well as two state-of-the-art scalable memory allocators for multithreaded programs: Hoard (**?**), and McRT (Hudson et al., 2006). These memory allocators use a combination of features to provide improved scalability: heap organizations that minimize contention between threads; minimizing false-sharing of heap-allocated objects, and bounding the amount of synchronization incurred by memory allocation.

We ran our microbenchmark on the 16-core Barcelona machine described in the previous section. We varied the number of allocations from ten thousand per thread to one hundred million per thread. Figure 7.9 shows the results of this experiment. The benchmark executes roughly 8 times faster using the PROMETHEUS thread-local free lists than the default `glibc` allocator. Furthermore, despite the advantages of the scalable memory allocators, they yield only marginal improvement for this particular allocation pattern.

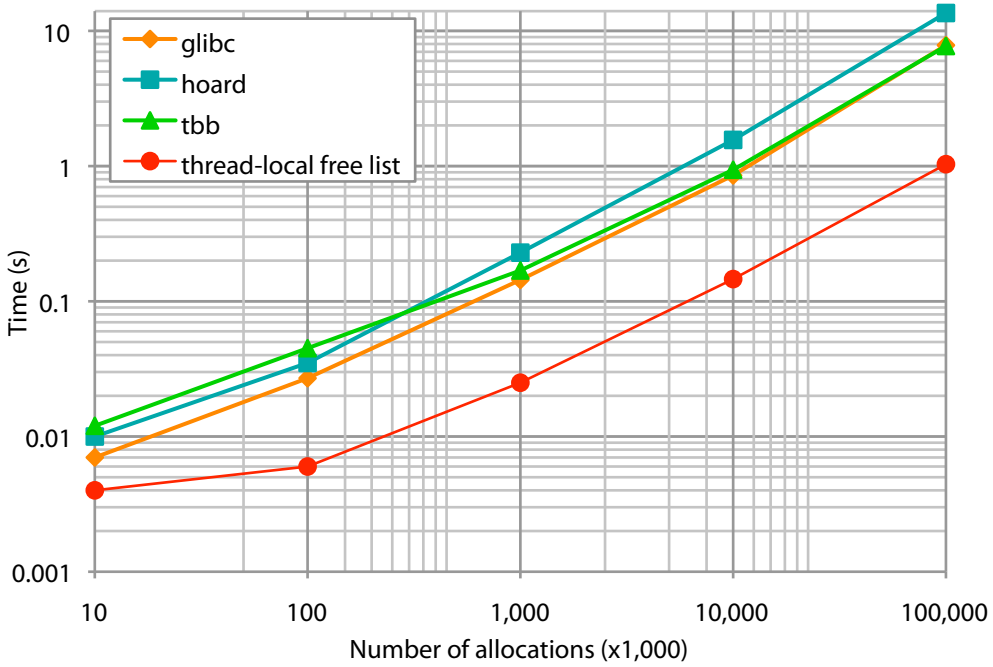


Figure 7.9: Performance of dynamic memory allocation vs. thread-local free lists

Performance Comparison of Control-Driven and Data-Driven Decomposition

To compare the performance of the pthread and PROMETHEUS versions of our programs, we calculate speedup relative to a sequential execution of the original program. Table 7.5 summarizes the results for the Barcelona machines, and Table 7.6 summarizes the results for the Nehalem machines. For each processor and benchmark, we give the time for the sequential execution of the original benchmark (BL), and our C++ version (C++), and the execution time for the parallel pthreads version and PROMETHEUS version (PR). The next two columns give the speedups of the pthread and PROMETHEUS parallel execution time over the sequential execution of the original program (BL). The last column gives the speedup of the PROMETHEUS version over our C++ program. Overall, the performance of the two models is quite similar, indicating that data-driven decomposition does not require compromising performance for these programs.

The PROMETHEUS implementation of `barnes-hut` lags behind its multi-threaded counterparts, especially on the 16-core Barcelona machine. We will

		Time (s)				BL Speedup		C++ Speedup
System	benchmark	BL	C++	PT	PR	PT	PR	PR
Barcelona (4)	barnes-hut	512.5	574.8	117.8	137.2	4.3	3.7	4.2
	black-scholes	309.0	302.5	104.5	81.0	3.0	3.8	3.7
	bzip2	281.0	282.0	77.6	78.9	3.6	3.6	3.6
	canneal	215.4	220.7	85.9	102.7	2.5	2.1	2.1
	dedup	45.1	39.5	13.0	12.6	3.5	3.6	3.1
	histogram	2.72	2.71	0.74	0.78	3.7	3.5	3.5
	reverse_index	46.8	23.3	8.7	6.6	5.6	7.3	3.5
	word_count	20.7	11.2	4.4	2.5	4.7	8.2	4.5
Barcelona (16)	barnes-hut	734.8	824.4	50.1	58.9	14.7	12.5	14.0
	black-scholes	369.1	359.1	68.8	72.6	5.4	5.1	4.9
	bzip2	334.7	334.1	26.7	28.8	12.5	11.7	11.6
	canneal	334.9	346.7	76.1	103.1	4.4	3.2	3.4
	dedup	61.2	52.4	15.6	13.6	3.9	4.5	3.9
	histogram	3.52	3.55	1.21	1.46	2.9	2.4	2.4
	reverse_index	63.4	31.0	6.0	5.3	10.6	12.1	5.9
	word_count	28.7	16.7	1.6	1.7	17.9	16.9	9.8
Barcelona (32)	barnes-hut	647.9	766.1	41.8	44.4	15.5	14.6	17.3
	black-scholes	322.1	312.4	52.2	90.0	6.2	3.6	3.5
	bzip2	308.2	310.6	17.6	15.8	17.5	19.5	19.7
	canneal	325.1	336.4	88.4	117.4	3.7	2.8	2.9
	dedup	55.1	48.1	20.5	15.4	3.6	4.6	3.1
	histogram	3.23	3.23	3.77	3.69	0.9	0.9	0.9
	reverse_index	65.0	27.6	14.1	6.8	4.6	9.6	4.1
	word_count	24.8	14.1	1.2	2.1	21.4	11.6	6.6

Table 7.5: Performance of pthreads and PROMETHEUS on Barcelona machines

		Time (s)				BL Speedup		C++ Speedup
System	benchmark	BL	C++	PT	PR	PT	PR	PR
Nehalem (4x2)	barnes-hut	280.2	322.3	61.3	67.6	4.6	4.1	4.8
	black-scholes	193.5	184.1	51.7	37.7	3.7	5.1	4.9
	bzip2	163.5	163.7	34.9	35.9	4.7	4.6	4.6
	canneal	135.2	137.0	46.5	44.0	2.9	3.1	3.1
	dedup	33.8	28.0	7.6	7.0	4.4	4.8	4.0
	histogram	1.41	1.33	0.28	0.28	5.0	5.0	4.8
	reverse_index	24.7	15.5	5.0	4.5	4.9	5.5	3.4
	word_count	13.5	7.7	2.6	1.5	5.2	8.8	5.1
Nehalem (8x2)	barnes-hut	316.1	357.5	37.8	41.4	8.3	7.6	8.6
	black-scholes	232.7	220.1	44.4	48.5	5.2	4.8	4.5
	bzip2	194.6	195.1	22.1	23.2	8.8	8.4	8.4
	canneal	123.0	124.1	44.9	45.8	2.7	2.9	2.7
	dedup	37.8	33.4	6.1	6.9	6.5	5.5	4.8
	histogram	1.58	1.60	0.19	0.19	8.3	8.3	8.4
	reverse_index	29.1	17.6	5.0	3.4	5.8	8.6	5.2
	word_count	13.5	7.7	2.6	1.5	5.2	9	5.1

Table 7.6: Performance of pthreads and PROMETHEUS on Nehalem machines

show in later results that the PROMETHEUS `foreach` construct for divide-and-conquer loops significantly improves the performance of `barnes-hut` over explicit delegation. While it is unlikely that these loops will be able to match the performance of multithreading on highly independent data-parallel programs like `barnes-hut`, this result indicates that our `foreach` loop might benefit from further optimization.

Running on the 16-core and 32-core Barcelona machines, `histogram` achieves a lower speedup than on any other system, including the similar 4-core Barcelona machine. We will see in later results that `histogram` achieves parallel speedups up to a certain number of worker threads, and then experiences a drastic drop-off in performance. This phenomenon does not appear when `histogram` runs on the Nehalem machines.

Our results show that our C++ implementation of `reverse_index` and `word_count` are roughly twice as fast as sequential execution of the original bench-

mark. This performance difference is largely caused by the different ways these programs store and aggregate the data produced by their parallel operations. The Phoenix versions of `reverse_index` and `word_count` maintain track links and word counts, respectively, in thread-private arrays. These lists are kept sorted, which incurs a high overhead for insertions, but allows for very efficient merging of the arrays. During a merge of two arrays, only the first element of each is examined at each step, so neither array need be searched. Sequential execution of these programs must pay the high insertion cost, which is not offset by the efficiency of merging since there is only one thread.

The PROMETHEUS versions of these programs use reducible containers to aggregate the results of computations stored in different objects. These containers allow for much faster insertions, but merging them via reduction is more expensive because while one container is traversed, the other container must be searched for matches. This trade-off is evident in the fact that the performance gap between the PROMETHEUS and Phoenix pthreads versions is smaller for machines with larger numbers of cores.

Sensitivity to Input Size

Figure 7.10 shows the performance of each benchmark running with small, medium, and large inputs. Note that for these measurements we compute speedup relative to the sequential PROMETHEUS program, rather than the original program. There is no basis for comparison between different benchmarks—the sizes of these inputs were determined separately. However, this data does allow us to see the sensitivity of each individual benchmark to input size. While most of the programs benefit from larger inputs, `barnes-hut` and `reverse_index` require a large amount of data to fully realize their potential.

The performance of `dedup` seems to be an exception to the relationship between performance and input size—it achieves its best performance for the medium input size. However, this result is misleading, because this input actually results in the program doing more work, evident in the fact that the output has a much higher compression ratio than the small and large input files.

Performance Scalability

Figures 7.11, 7.12, and 7.13 show the scalability of the PROMETHEUS benchmarks as a function of the number of worker threads used by the runtime. Both `barnes-hut` and `bzip2` continue scaling to 16 workers and beyond, but the other programs

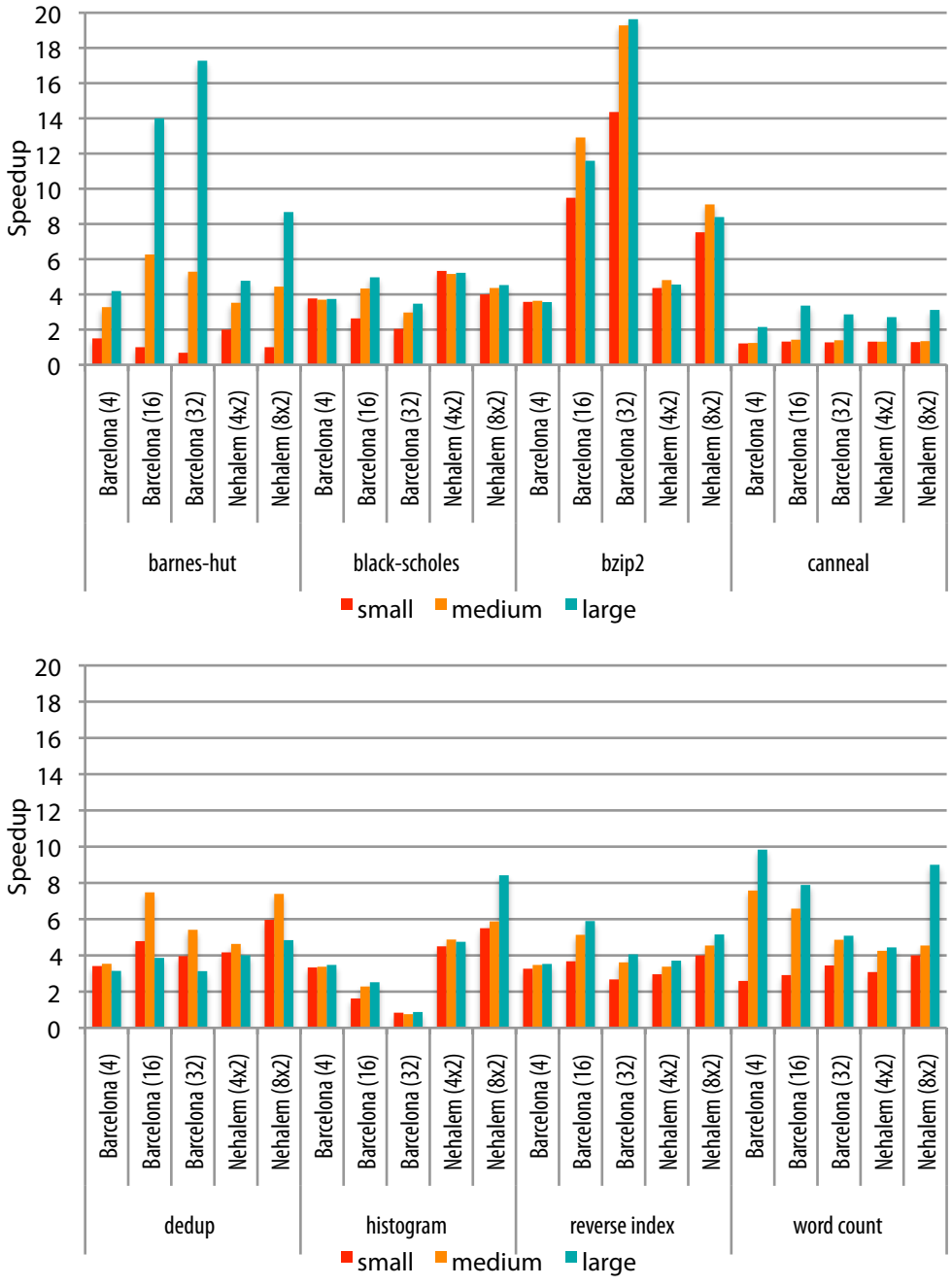


Figure 7.10: Benchmark performance by input size

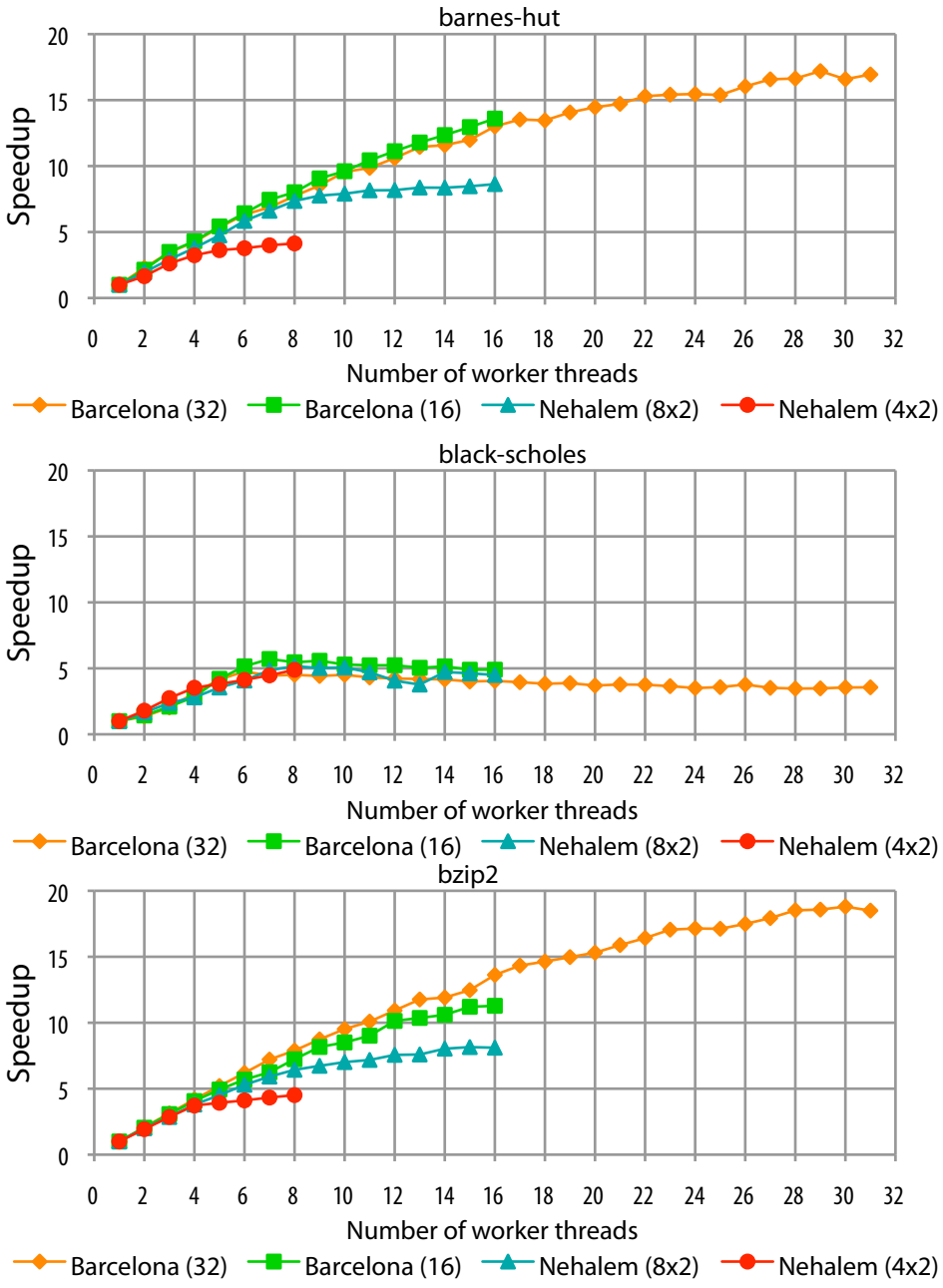


Figure 7.11: PROMETHEUS scalability for `barnes-hut`, `black-scholes`, and `bzip2`

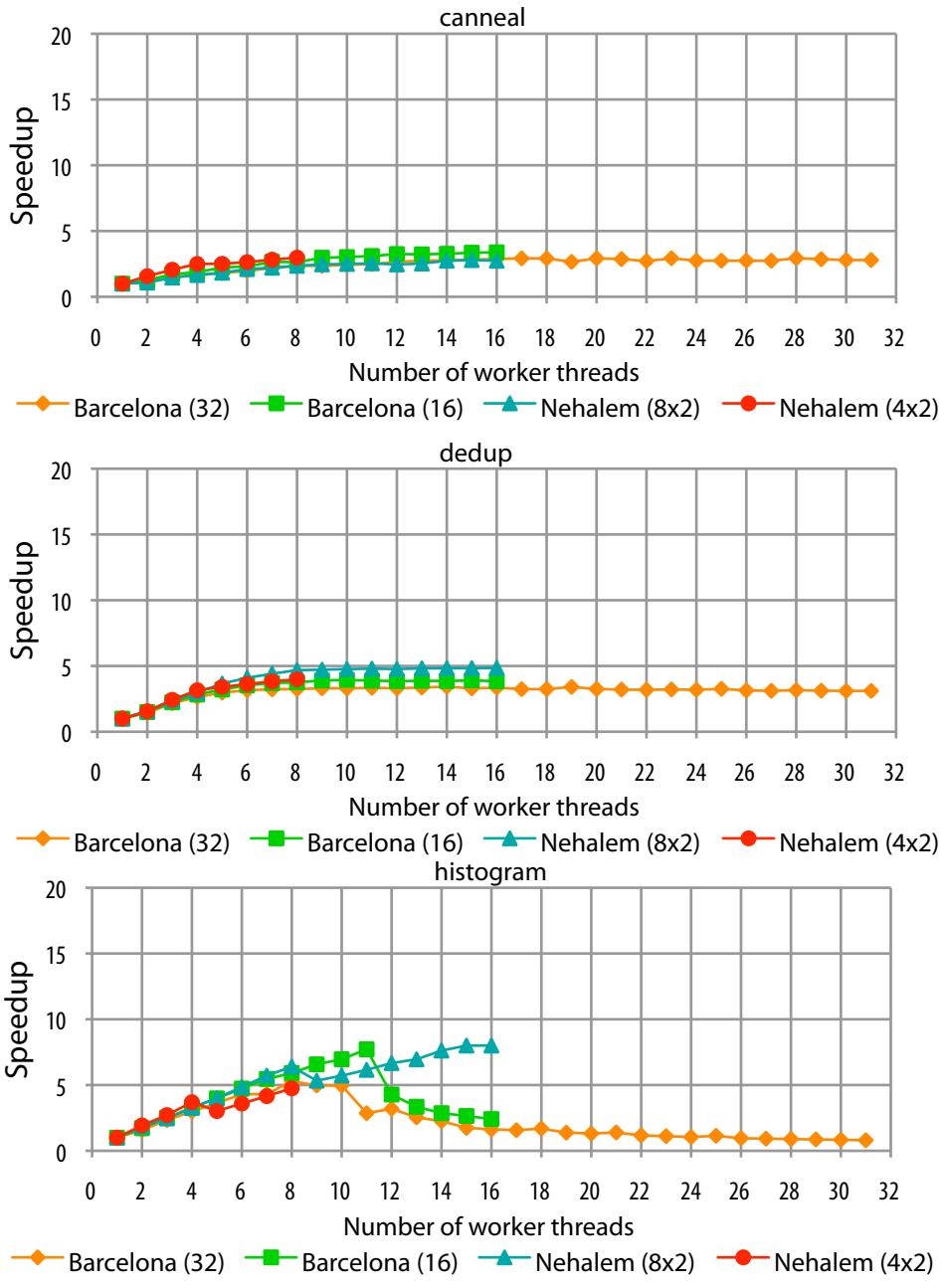


Figure 7.12: PROMETHEUS scalability for `canneal`, `dedup`, and `histogram`

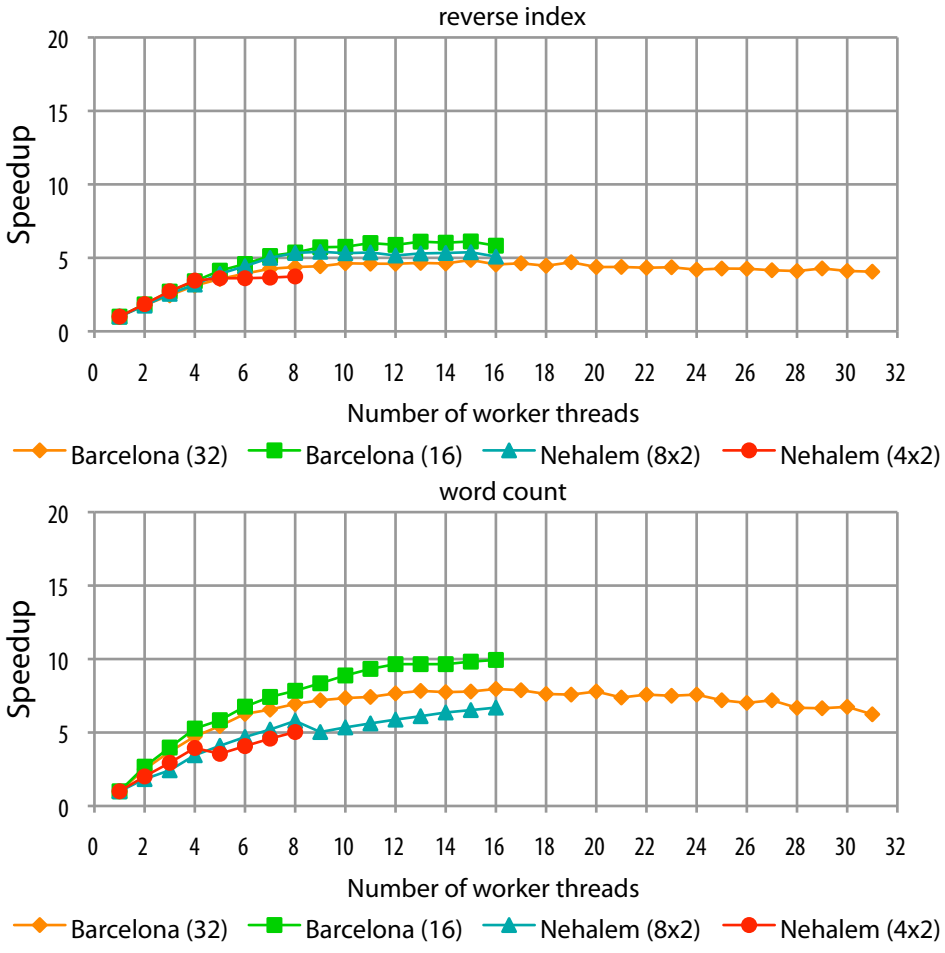


Figure 7.13: PROMETHEUS scalability for `reverse_index` and `word_count`

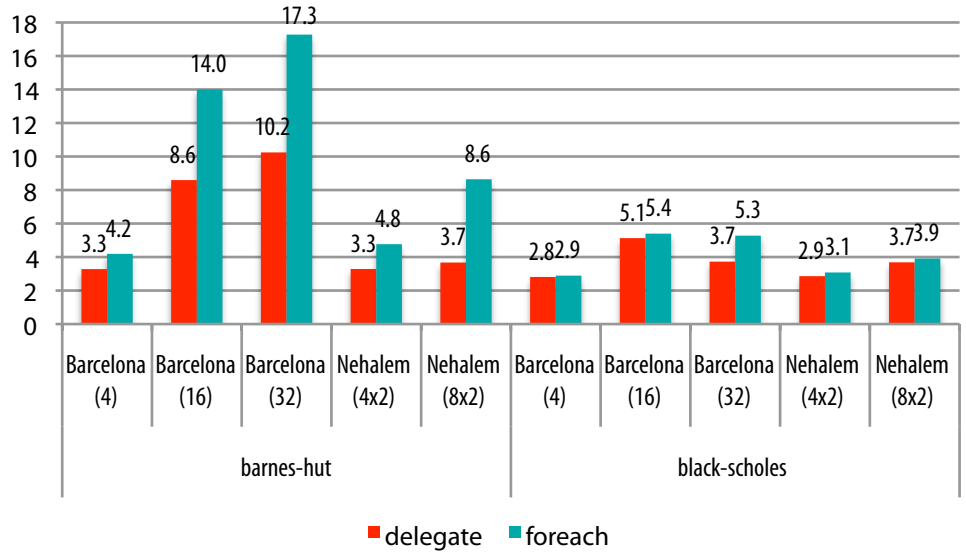


Figure 7.14: Performance of the `foreach` loop

saturate much earlier.

The `histogram` benchmark shows a peculiar scaling trend on the Barcelona servers—scaling up to a certain number of threads above which the execution time of the program increases drastically. Because the pthreads version of the program also exhibits this behavior, we do not believe it indicates a problem with PROMETHEUS. We also note that this problem does not occur on the Nehalem systems.

Parallel Loop Performance

To illustrate the importance of efficient parallel loops, we measured the performance two loop-based benchmarks using both explicit delegation and the `foreach` loop. We compared our standard `foreach`-based implementation of `barnes-hut` with a version that performs explicit delegation inside of a standard `for` loop. We also modified `black-scholes` to use loops instead of the pipeline-style implementation described in Section 7.1. These versions do not perform as well, but they serve to illustrate the difference between fine- and coarse-grained loops.

We present the results of this comparison in Figure 7.14 . For `black-scholes`, which exhibits very coarse-grained parallelism, the `foreach` loop provides only

Program	Invocations (mean)	Allocated stacklets (mean)	Allocated stacklets (percentage)
barnes-hut	7499925	632.8	< 0.01%
black-scholes	20000000	17	< 0.01%
bzip2	1566	17	0.01%
canneal	9000000	17	< 0.01%
dedup	167609	17	0.01%
histogram	16	17	106.3%
reverse_index	78371	17	0.02%
word_count	17	63	27.0%

Table 7.7: Stacklet recycling effectiveness

marginal benefit because even when using explicit delegation, distribution of the loop iterations is not on the critical path. By contrast, for the finer-grained parallelism in `barnes-hut`, the `foreach` loops result in substantial performance improvement: on the 16-core Barcelona system, `foreach` improves a speedup of 8.6 to 14.0; on the 32-core Barcelona system, `foreach` improves a speedup of 10.2 to 17.3; on the 8-core, 16-thread Nehalem system, `foreach` more than doubles the speedup of 3.7 to 8.6.

Effectiveness of the Stacklet Recycling Mechanism

In Chapter 5, we argued that allocation and deallocation of stacklets could be performed efficiently by recycling previously allocated stacklets, avoiding an expensive `mmap`. system call. The PROMETHEUS runtime tracks both the number of delegated method invocations (and tasks spawned for `foreach` loops), as well as the number of stacklets allocated using `mmap`. This data can be used to determine the effectiveness of stacklet recycling by calculating the number of delegated or spawned invocations that require allocation.

We collected the stacklet allocation data for ten runs of our benchmark programs on the 16-core Barcelona machine, and we present the results in Table 7.7. Except for `barnes-hut`, all of the other programs require only 17 stacklet allocations—one for each worker thread, and one additional stacklet for the scheduling context of the original program thread. As a result, most of the bench-

mark programs require stacklet allocation for a very small fraction of all delegated or spawned method invocations. The `foreach` loop construct used in `barnes-hut` requires many additional stacklets for the recursive divide-and-conquer of the loop range, and thus requires many more stacklets, but it also requires stacklet allocation very infrequently. The exceptions are `histogram` and `word_count` which exhibit very coarse granularities of parallelism, resulting in low numbers of delegated method invocations. In particular, `histogram` delegates only a single method invocation for each worker thread, so every delegation results in stacklet allocation. Overall, the stacklet recycling mechanism is highly effective at minimizing the number of `mmap` calls.

7.4 SUMMARY

We began this chapter by describing the benchmarks used in our evaluation, including a description of how we applied PROMETHEUS to facilitate data-driven decomposition. After presenting our experimental methodology, we presented the results of our evaluation. Based on these results, we observe that for the programs we studied, data-driven decomposition was able to achieve performance competitive with that of a control-driven decomposition. We examined the sensitivity of these programs to input size and showed how they scale to increasing thread counts. Finally, we showed that the PROMETHEUS `foreach` construct is essential to extracting fine-grained loop parallelism.

8 CONCLUSIONS AND FUTURE WORK

There must be a beginning of any great matter, but the continuing unto the end until it be thoroughly finished yields the true glory.

— Sir Francis Drake (n.d.)

In this dissertation, we proposed *data-driven decomposition*, a new parallel execution model that (1) dynamically decomposes a sequential program based on the data manipulated by each operation, (2) preserves the sequential ordering of operations on a particular set of data, and (3) parallelizes execution of operations on disjoint sets of data. The thesis of this dissertation is that data-driven decomposition yields repeatable, predictable, and efficient parallel execution. To support this thesis, we presented the following contributions:

1. A definition of *sequential determinacy*—the property that each variable is assigned the same sequence of values as a sequential execution in any parallel execution with a given input—augmenting the repeatability of *determinacy* (Karp and Miller, 1966) with the predictability of sequential execution (in Chapter 2);

2. Constructs for data-driven decomposition: the *private object*, a distinct collection of data that may only be manipulated by the operations defined by its specification; and the *serializer*, which preserves the sequential ordering of method invocations on a private object, but executes these methods asynchronously with respect to other operations on other objects (in Chapter 3);

3. A program representation based on annotations to traditional sequential, object-oriented programming constructs such as class specifications and method invocations (in Chapter 3);

4. A serializer implementation that exploits state-of-the-art dynamic task scheduling algorithms to execute method invocations delegated to different serializers; and a technique for *dynamic task extension* to ensure the sequential ordering of method invocations delegated to the same serializer (in Chapter 4);

5. The PROMETHEUS runtime, which uses a combination of fast user-level context switching and a stacklet-based activation record management to provide the first library for dynamic scheduling via unrestricted work stealing with lazy task creation (in Chapter 5);

6. Recognition of the *receiver identification problem*, which restricts any operation in a determinate program from manipulating its receiver until all previous operations have identified their receivers; and the proposal for *receiver disambiguation* to overcome the limitations of sequential receiver identification on parallelism (in Chapter 6).

In addition to these contributions, performing the work described in this dissertation yielded several other insights about writing software intended for parallel execution. We review these in the next section.

8.1 INSIGHTS AND IMPLICATIONS

In our experience so far, data-driven decomposition has met its original goals, allowing programmers to develop parallel software using the familiar sequential program representation. The repeatability and predictability afforded by sequential determinacy means that most testing and debugging can be performed on a sequential execution. This represents a substantial improvement over the difficult process of identifying, reproducing, and correcting bugs in multithreaded programs. However, data-driven decomposition is not without limitations, and significant barriers to realizing parallel execution remain.

Limitations of Data-Driven Decomposition

During the initial development of data-driven decomposition, we thought that restricting parallel execution to operations on disjoint objects might limit the expressive power of the model in comparison with control-driven decomposition. After performing the research documented in this dissertation, we now believe that this model imposes very few limitations on the kinds of computations that can be expressed. The reasons for this are twofold: First, as we discussed in Chapter 3, the tenets of modular and object-oriented programming include practices such as encapsulation and ownership, which result in the same organization of data required for data-driven decomposition. Second, restricting manipulation of a set of data to at most a single operation should not be any more of a limitation for data-driven decomposition than it is for control-driven decomposition. While multithreading provides the abstraction of shared memory, programs must use synchronization to ensure that only one operation modifies a particular set of data at any given time.

The remaining question is then whether the organization of data required for

data-driven decomposition limits the expressive power of the model. We believe that it does not. Nested delegation allows each set of data to be partitioned into subsets that can be operated on independently, and multiple delegation allows a single operation to operate on multiple objects at once. Composing these techniques allows the programmer to express computation on any combination of data in the program.

Instead, the primary limitation of data-driven decomposition seems to be the requirement of a specific ordering of operations on each set of data. This leads to the receiver identification problem discussed in Chapter 6, which prevents executing an operation until the receivers of all earlier-ordered operations are known. Receiver disambiguation partially alleviates this problem by facilitating parallel receiver identification. However, it appears that fully overcoming this limitation will likely require speculative techniques that allow an operation to immediately modify its receiver, rolling back and re-executing in the event that an earlier operation identifies the same receiver.

Remaining Challenges for Parallel Software Development

Despite the benefits of data-driven decomposition, parallel programs remain significantly more difficult to develop than sequential programs. Using a sequential representation does not mean that an existing program can be parallelized by simply annotating classes and methods. Programmers must identify or develop a suitable parallel algorithm to perform the desired computation, and these algorithms are almost always significantly more complex than an equivalent sequential algorithm. Programmers must also pay careful attention to the organization and structure of the data processed by the program, which can have a significant impact on the parallelism and locality. We therefore believe that many current applications will require a significant amount of redesign and reimplementation to fully exploit multi-core processors.

Data-driven decomposition greatly simplifies debugging program correctness, but debugging performance remains a difficult problem. Identifying the most computationally intensive parts of a program is straightforward using existing profiling tools. While this allows the programmer to target parallelization efforts at performance bottlenecks, it yields no information about the parallelism of these computations. This situation is gradually improving, thanks to tools such as HPCTOOLKIT (**?**) and Cilkview (**?**), which measure the work, span (critical path), and the resulting parallelism of a program. Further improvement of these techniques and development of new tools for performance analysis are imperative

to improving the accessibility of parallel software development.

Cache effects cause some of the most vexing performance problems, and can inhibit the performance of programs that are otherwise highly parallel. These problems can be very difficult to diagnose, since they are not a direct consequence of the computations in the program. Our experience with tuning parallel execution has led us to recognize highly erratic performance as an indicator of cache lines thrashing between processor cores due to false sharing. Tools like Intel VTune (**?**), which provide access to cache miss performance counters, are invaluable for diagnosing the source of these conflicts. Ultimately, we have adopted the defensive strategy of aligning any frequently accessed data structure on a cache-line boundary. Unfortunately, specifying data alignment is not supported by most current programming languages, and instead requires obscure, compiler-specific attributes and intrinsic functions. Combined with the need to explicitly encode cache-line sizes, this results in code that is non-portable and difficult to write and maintain. Furthermore, these annotations do not produce the desired outcome for language constructs as simple as arrays. These problems have forced us to implement custom alignment templates and memory allocation routines to perform the work in this dissertation. In the future, languages should provide abstract, machine-independent specification of alignment to ameliorate these difficulties.

8.2 FUTURE WORK

During the development of data-driven decomposition and the preparation of this dissertation, we have identified several opportunities for future research. One clear immediate direction is the application of data-driven decomposition to a broader range of applications to demonstrate its applicability and identify any shortcomings of the programming interface and execution model. Additionally, tools for debugging and performance will be an important part of supporting large-scale software development with data-driven decomposition. As we described in Chapter 3, an important goal of these tools should be detecting safety violations caused by inadvertently shared data that introduces determinacy races in the program. Extending existing dynamic determinacy race detection algorithms, such as those proposed by Feng and Leiserson (1997) and Bender et al. (2004), would be an important step in this direction. While these can detect incorrect execution, developing sound static analyses to check the safety of data-driven decomposition would ensure that any execution of a program was correct.

In our study of data-driven decomposition, we have observed cases where

ensuring determinacy is complicated by non-determinism in sequential programs. The most obvious example is random number generation, although this can be avoided by using a deterministic seed. We have identified two sources of non-determinism that are more problematic. The first source of non-determinism is dynamic memory allocation, which generally returns different addresses for the allocation of a particular object in different runs of the program. Usually this has no obvious impact on program execution, but it will occasionally manifest when these addresses are used for operations like hashing or sorting. The second source of non-determinism in sequential programs is time. Any time control flow is affected by the duration of some event, the result is likely to be nondeterministic. Future research on determinate execution models should investigate how to address such sources of non-determinism.

Our current implementation of data-driven decomposition enforces the sequential ordering of all method invocations on a particular object. However, it is worth considering whether there are cases when it may be beneficial to relax this ordering. Commutativity annotations, such as those proposed by **?**, **?**, and Bocchino et al. (2009b), could be used to identify methods that can be reordered without changing the final state of the object. This provides the programmer a mechanism to judiciously introduce nondeterminacy into the program, and may be necessary to overcome the receiver identification problem for some algorithms. Allowing reordering of commutative operations in a data-driven decomposition raises the following questions: (1) In what situations does such reordering significantly enhance performance? (2) How should commutativity be expressed in the program? (3) How can we introduce nondeterminacy selectively, so as not to compromise the determinacy of other operations?

One avenue for future research is exploring whether dynamic parallelization can achieve better performance than static parallelization. We believe that it can, especially for irregular programs with unstructured parallelism. A static, control-driven decomposition of such programs can result in significant contention for locks. Threads that fail to acquire a lock may block, and if there is other independent work they could be doing, it may be unnecessarily delayed. Dynamic data-driven decomposition should be limited only by the data flow through the program. This should result in higher performance than a control-driven decomposition for programs that exhibit significant lock contention. Our initial work in this area has produced promising results, and we plan to investigate it more thoroughly in the future.

A related question is whether optimistic models, such as speculative multithreading or transactional memory, are able to extract higher performance from

some applications than a dynamic model. We conjecture that optimistic models would have an advantage for applications were most operations are truly independent, but the receiver identification problem limits a dynamic decomposition from fully exploiting this independence. If such applications can be identified, they would warrant investigation of how data-driven decomposition can be augmented with support for speculative execution.

Because data-driven decomposition results in parallel operations on disjoint data, it seems like a promising match for the emerging class of high throughput compute accelerators, including graphics processors (or GPUs), and heterogeneous processors like the IBM Cell. These accelerators provide arithmetic throughput dwarfing that of current microprocessors, and can potentially benefit many computationally intensive calculations. However, the programming models for these accelerators require intimate knowledge of the underlying hardware and provide only a minimal amount of abstraction. If runtime support for data-driven decomposition could be retargeted to these accelerators, it would allow programmers to exploit their computational horsepower at a much higher level of abstraction.

Given that determinate parallel execution models provide significant benefits to software developers raises the question of whether it might also benefit hardware. **?** observe that because determinate models provide information about the data accessed by each parallel operation and prohibit data races, they provide opportunities for simplifying cache coherence and memory consistency models, as well as reducing interprocessor communication. We agree with their conclusions, and observe that such simplifications will likely be necessary in the future as processors scale to increasingly numbers of cores. If determinate programming models facilitate simpler, more scalable, and higher performance hardware, it would make them even more compelling.

8.3 CONCLUSIONS

The development of data-driven decomposition presented in this dissertation leads us to the following set of conclusions. First, parallelism is all about data. As Bernstein observed in 1966, achieving parallel execution requires identification of operations on independent data. Data should therefore be the primary focus of future parallel execution models. Second, the identity of the data manipulated by each operation can be identified and exploited dynamically. The paramount importance of data to parallelism thus gives dynamic decomposition a fundamental advantage over static decomposition. Third, ordering is the key to simplifying parallel programming. It should be specified by the programmer and then enforced,

rather than recklessly discarded, by the execution model. The intuitive nature and widespread use of the sequential programming model make it the logical basis for determining the ordering of program operations.

The work described in this dissertation represents our efforts to apply these observations. While some aspects of our implementation of data-driven decomposition may prove to be flawed or misguided, we believe these principles will be instrumental in developing accessible, reliable, and efficient parallel execution models.

A OVERVIEW OF C++ TEMPLATES

Software frequently requires components with similar functionality that operate on different types. For example, it is useful to have a function that computes the maximum of two numbers, whether those numbers are integral or floating-point. Likewise, containers such as a binary tree or a hash table are useful for any number of different types. Ideally, a programmer should only express such functionality once, in a generic way, rather than duplicating it for every type.

C++ addresses this need with templates, a powerful language feature that allow programmers to abstract concrete types out of classes and functions to create generic versions. The ability to write generic code that can operate on different concrete types in different contexts is often called *generic programming* (Stroustrup, 1997, ch.13) or *parametric polymorphism* (Pierce, 2002, pg. 319). Template classes and functions are parameterized on types, and these type variables are evaluated when the program is compiled. Whenever a generic class or function is used in the program, the compiler *instantiates* it by replacing the type variables with the appropriate concrete types, and specializes the resulting code for those types. Once a template is instantiated, the compiler performs the same type checking on these classes and functions that it would on non-template code.

A.1 TEMPLATE CLASSES

We illustrate the usage of C++ templates with several simple examples. Figure A.1 lists the code for an example C++ template class. The `pair_t` class is a simple generic container for holding two values. The class declaration is preceded by the `template` keyword on line 1, which is parameterizes the class on two types, `T1` and `T2`. These parameters determine the types of the two values stored in the `pair_t` container (lines 4 and 5), and are used in every method signature that refers to the types of these values—the constructor (line 8), the accessor for each value (lines 12 and 13), and the mutator for each value (lines 14 and 15).

The programmer explicitly instantiates a template class by specifying the concrete types to be substituted for the type parameters. In our example, the programmer declares a `pair_t` object to hold integer and floating-point values. The statement `pair_t<int, float>` (line 19) causes the compiler to instantiate the `pair_t` class and substitute `int` for `T1` and `float` for `T2`.

```
1  template <typename T1, typename T2>
2  class pair_t {
3  private:
4      T1 first;
5      T2 second;
6
7  public:
8      pair_t (T1 first, T2 second) :
9          first (first), second (second)
10     {}
11
12     T1 get_first () const { return first; }
13     T2 get_second () const { return second; }
14     void set_first (T1 first) { this->first = first; }
15     void set_second (T2 second) { this->second = second; }
16 };
17
18 int main () {
19     pair_t <int, float> pair (42, 3.14159); // explicit instantiation
20     cout << "first=" << pair.get_first ()
21          << ", second=" << pair.get_second ()
22          << endl;
23 }
```

Figure A.1: Example of a C++ template class

A.2 TEMPLATE FUNCTIONS

Figure A.2 gives a simple template function. The `swap` function exchanges the values stored in two variables. This function is parameterized on the type `T` (line 1), representing the type of the values being swapped. The function takes two arguments of the parameter type `T` by reference (line 2), and exchanges their values (lines 3–5).

In contrast with template classes, which must be explicitly instantiated, template functions can be implicitly instantiated when the compiler can infer the types. The call to `swap` on line 11 passes in two integer arguments, causing an implicit instantiation of `swap` with `int` substituted for `T`. In some cases, the compiler cannot infer the type parameters from the context of the program—for example, when the function arguments are also parameterized types—and the programmer must explicitly instantiate the template function, as shown on line 12.

```
template <typename T>
void swap (T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main () {
    int x = 3;
    int y = 6;
    swap (x, y); // implicit instantiation
    swap <int> (x, y); // explicit instantiation
    cout << "x=" << x << ", y=" << y << endl;
}
```

Figure A.2: Example of a C++ template function

```
// Template class
template <int num>
class factorial_t {
public:
    static const int value = num * factorial_t <num - 1>::value;
};

// Template specialization for num=0
template <>
class factorial_t <0> {
public:
    static const int value = 1;
};

int main () {
    int x = factorial_t <3>::value; // evaluates to 6
    int y = factorial_t <4>::value; // evaluates to 24
    cout << "x=" << x << ", y=" << y << endl;
}
```

Figure A.3: Example of a C++ template metaprogram

A.3 TEMPLATE METAPROGRAMMING

Our examples so far have only hinted at the power of C++ templates. The compile-time evaluation performed for template instantiation enables *template metaprogramming*—the use of templates as a Turing-complete language for compile-time execution (Veldhuizen, 1995; Czarnecki and Eisenecker, 2000). Figure A.3 lists a simple template metaprogram for computing the factorial of an integer. This example utilizes two abilities of templates we have not yet discussed. First, template classes and functions can be parameterized on primitive types such as integers and booleans. Second, template classes and functions can be *specialized* for particular concrete types or primitive values. The compiler always prefers these specialized templates over generic ones, allowing the programmer to customize the behavior of the template for particular template parameter values.

The `factorial_t` class (lines 2–6) is parameterized on the integer `num` (line 2). This class has a single static field `value`, a constant that is initialized using the familiar recursive definition of the factorial function (line 5). Each instantiation of `factorial_t` for a specific value of `num` causes the instantiation of `factorial_t` for `num - 1`, and `value` is assigned the product of these two numbers.

Like any recursive computation, our factorial metaprogram must have a base case that terminates the recursion. To this end, `factorial_t` class template is specialized for the case when `num` is zero on lines 9–13. This class sets the `value` field to one (line 12).

The programmer can calculate the factorial of any positive integer using the `factorial_t` template class. On line 16, the programmer assigns the value of `factorial_t<3>` to the variable `x`, which will be evaluate to 6, and on line 17, the programmer assigns the value of `factorial_t<4>` to the variable `y`, which will evaluate to 24. These values are evaluated at compile time—not run time—by the recursive instantiation of the `factorial_t` template, and become constants in the program.

DISCARD THIS PAGE

COLOPHON

This document is set in Adobe Minion Pro 12 point; the sans-serif face is Adobe Myriad Pro, and the mathematical and symbol font is AMS Euler. It was set by the author on a Macintosh using pdfTeX and William C. Benton's University of Wisconsin dissertation template, available at `http://github.com/willb/wi-thesis-template`. This template uses the `microtype` package, the `memoir` class, a locally modified version of the `bringhurst` chapter style, and the `natbib` bibliography package.

REFERENCES

Acar, Umut A., Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the 12th annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1–12.

Allen, Eric, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele, Jr. 2007. Project Fortress: a multicore language for multicore processors. *Linux Magazine*.

Allen, Matthew D., Srinath Sridharan, and Gurindar S. Sohi. 2009. Serialization sets: a dynamic dependence-based parallel execution model. In *Proceedings of the 14th symposium on Principles and Practice of Parallel Programming*, 85–96.

Almeida, Paulo Sérgio. 1997. Balloon types: controlling sharing of state in data types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 32–59.

Amdahl, Gene M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 483–485.

Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo. 1967. The IBM System/360 Model 91: machine philosophy and instruction-handling. *IBM Journal of Research and Development* 8–24.

Apple. 2009. Grand central dispatch. `http://images.apple.com/jp/macosx/technology/docs/GrandCentral_TB_brief_20090608.pdf`.

Armstrong, Deborah J. 2006. The quarks of object-oriented development. *Communications of the ACM* 49(2):123–128.

Arora, Nimar S., Robert D. Blumofe, and Greg C. Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 119–129.

Artho, Cyrille, Klaus Havelund, and Armin Biere. 2003. High-level data races. In *Proceedings of the Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, 207–227.

Baker, Henry C., Jr., and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, 55–59.

Bender, Michael A., Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the 16th annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 133–144.

Benton, William C., and Charles N. Fischer. 2009. Mostly-functional behavior in Java programs. In *Proceedings of the 10th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 29–43.

Bergan, Tom, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th annual conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 53–64.

Berger, Emery D., Ting Yang, Tongping Lieu, and Gene Novark. 2009. Grace: safe multithreaded programming for C/C++. In *Proceedings of the 24th conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 81–96.

Bernstein, A. J. 1966. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers* EC-15(5):757–763.

Bienia, Christian, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 72–81.

Blelloch, Guy E. 1996. Programming parallel algorithms. *Communications of the ACM* 39(3):85–97.

———. 2009a. Is parallel programming hard? `http://software.intel.com/en-us/articles/is-parallel-programming-hard/`.

———. 2009b. Parallel thinking. In *Proceedings of the 14th symposium on Principles and Practice of Parallel Programming*, 1–2. Keynote address.

Blelloch, Guy E., and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* 8(2):119–134.

Blumofe, Robert D., Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth annual symposium on Principles and Practice of Parallel Programming (PPoPP)*, 207–216.

Blumofe, Robert D., and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46(5):720–748.

Bocchino, Robert L., Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009a. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar)*.

Bocchino, Robert L., Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Moshen Vakilian. 2009b. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 97–116.

Boland, L. J., G. D. Granito, A. U. Marcotte, B. U. Messina, and J. W. Smith. 1967. The IBM System/360 Model 91: storage system. *IBM Journal of Research and Development* 54–68.

Boyapati, Chandrasekhar, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *Proceedings of the 30th symposium on Principles Of Programming Languages (POPL)*, 213–223.

Burton, F. Warren, and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Proceedings of the conference on Functional Programming languages and Compueter Architecture (FPCA)*, 187–194.

Cardelli, Luca. 2004. Type systems. In *CRC Handbook of Computer Science and Engineering*. 2nd Edition, CRC Press.

Chamberlain, Bradford L., David Callahan, and Hans P. Zima. 2007. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications (IJHPCA)* 21(3):291–312.

Chambers, Craig, and Weimin Chen. 1999. Efficient multiple and predicated dispatch. In *Proceedings of the 14th conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 238–255.

Charles, Philippe, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 519–538.

Chase, David, and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the 17th annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 21–28.

Cilk Arts. 2008. Cilk++ programmer's guide.

Clarke, Dave, and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th conference on Object-Oriented Programming, Systems, Languages, and Applications*, 292–310.

Clifton, Curtis, Todd Millstein, Gary T. Leavens, and Craig Chambers. 2006. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28(3):517–575.

Coffman, E. G., Jr., M. J. Elphick, and A. Shoshani. 1971. System deadlocks. *ACM Computing Surveys* 3(2):67–78.

Compaq. 2002. *Alpha architecture reference manual*.

Contreras, Gilberto, and Margaret Martonosi. 2008. Characterizing and improving the performance of Intel Threading Building Blocks. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 57–66.

Conway, Pat, and Bill Hughes. 2007. The AMD Opteron northbridge architecture. *IEEE Micro* 27(2):10–21.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT Press.

Czarnecki, Krysztof, and Ulrich W. Eisenecker. 2000. *Generative programming: Methods, tools, and applications*. Addison-Wesley.

Denning, Peter J., and Jack B. Dennis. 2010. The resurgence of parallelism. *Communications of the ACM* 53(6):30–32.

Devietti, Joseph, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 85–96.

Dijkstra, E. W. 1983. Solution of a problem in concurrent programming control. *Communications of the ACM* 26(1):21–22.

Dijkstra, Edsger W. 1972. *Notes on structured programming*, chap. 1. Academic Press Ltd.

Duran, Alejandro, Roger Ferrer, Eduard Ayguadeé, Rosa M. Badia, and Jesus Labarta. 2009. A proposal to extend the openmp tasking model with dependent tasks. *International Journal of Parallel Programming* 37(3):292–305.

Eager, Derek L., John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers* 38(3):408–423.

Feng, Mingdong, and Charles E. Leiserson. 1997. Efficient detection of determinacy races in cilk programs. In *Proceedings of the ninth annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1–11.

Flood, Christine H., David Detlefs, Nir Shavit, and Xioalan Zhang. 2001. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the USENIX symposium on Java Virtual Machine research and technology (JVM)*.

Frigo, Matteo. 2009. The thorny problem of the cactus stack. `http://software.intel.com/en-us/articles/the-thorny-problem-of-the-cactus-stack/`.

Frigo, Matteo, Charles E. Leiserson, Harold Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *Proceedings of the annual symposium on Foundations Of Computer Science (FOCS)*, 287–297.

Frigo, Matteo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI)*, 212–223.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design patterns*. Addison-Wesley.

Garey, Michael R., and David S. Johnson. 1979. *Computers and intractability: A guide to the theory of np-completeness*. W. H. Freeman and Company.

Goldstein, Seth Copen, Klaus Erik Schauser, and David E. Culler. 1996. Lazy threads: implementing a fast parallel call. *Journal of Parallel and Distributed Computing* 37(1):5–20.

Gray, Jim. 1985. Why do computers stop and what can be done about it? Tech. Rep., Tandem TR 85.7.

Gregor, Douglas, Jaakko Järvi, Jens Maurer, and Jason Merrill. 2007. Proposed wording for variadic templates (revision 2). Tech. Rep. N2242=07-0102, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++.

Gscwind, Michael. 2007. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming* 35(3).

Guo, Yi, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, 1–12.

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

Hauck, E. A., and B. A. Dent. 1968. Burroughs' B6500/B7500 stack mechanism. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 245–251.

Heidelberger, Philip, Alan Norton, and John T. Robinson. 1990. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers* 39(1):133–138.

Hendler, Danny, and Nir Shavit. 2002. Work dealing. In *Proceedings of the 14th Symposium on Parallel Algorithms and Architectures (SPAA)*, 164–172.

Herlihy, Maurice. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(1):124–149.

Herlihy, Maurice, Victor Luchangco, and Mark Moir. 2003. Obstruction-free synchronization: double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, 522–529.

Herlihy, Maurice, and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

Herlihy, Maurice P., and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12(3):463–492.

Hewitt, Carl, and Peter Bishop Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the third International Joint Conference on Artificial Intelligence (IJCAI)*, 235–245.

Hillis, W. Daniel, and Guy L. Steele, Jr. 1986. Data parallel algorithms. *Communications of the ACM* 29(12):1170–1183.

Hoare, C. A. R. 1961a. Algorithm 63: Partition. *Communications of the ACM* 4(7):321.

———. 1961b. Algorithm 64: Quicksort. *Communications of the ACM* 4(7):321.

Hogg, John. 1991. Islands: aliasing protection in object-oriented languages. In *Proceedings of the sixth conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 271–285.

Hogg, John, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. 1992. The Geneva convention on the treatment of object aliasing. *OOPS Messenger* 3(2):11–16.

Hudson, Richard L., Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. 2006. McRT-Malloc: a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*, 74–83.

IBM. 1975. *System/370 principles of operation*. Order Number GA22-7000.

———. 2005. *PowerPC microprocessor family: The programming environments manual for 64-bit microprocessors (version 3.0)*.

IEEE. 2001. *Ieee standard 1003.1-2001*. IEEE and The Open Group.

Intel. 2009a. *Intel 64 and IA-32 architectures software developer's manual: Volume 2b: Instruction set referece, a–m*.

———. 2009b. An introduction to the Intel QuickPath interconnect. `http://www.intel.com/technology/quickpath/introduction.pdf`.

ISO/IEC. 2010. Working draft, standard for programming language C++. Tech. Rep. N3090=10-0080, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++.

Karp, Richard M., and Raymond E. Miller. 1966. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics* 14(6):1390–1411.

Knuth, Donald. 1997. *The art of computer programming*, vol. 1. 3rd ed. Addison-Wesley.

Kukanov, Alexey, and Michael J. Voss. 2007. The foundations for scalable multi-core software in Intel threading building blocks. *Intel Technology Journal*.

Lamport, Leslie. 1974. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM* 17(8):453–455.

———. 1983. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5(2):190–222.

Lavender, R. Greg, and Douglas C. Schmidt. 1995. Active:object: An object behavioral pattern for concurrent programming. In *Proceedings of the second conference on Pattern Languages of Programs (PLoP)*.

Lea, Doug. Concurrency JSR-166 interest site. `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

———. 2000. A Java fork/join framework. In *Proceedings of the Java Grande conference*, 36–43.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes* 31(3):1–38.

Lee, Edward. 2006. The problem with threads. *Computer* 39(5):33–42.

Leijen, Daan, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 227–242.

Leiserson, Charles E. 2009. The Cilk++ concurrency platform. In *Proceedings of the 46th annual Design Automation Conference (DAC)*, 522–527.

Lu, Shan, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 329–339.

Lucassen, John M., and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th symposium on Principles of Programming Languages (POPL)*, 47–57.

Lüling, Renhard, and Burkhard Monien. 1993. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the fifth annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 164–172.

Martin, Robert C. 1996. The open-closed principle. *C++ Report*.

McKenney, Paul E. 1996. Selecting locking primitives for parallel programming. *Communications of the ACM* 39(10):75–82.

Mellor-Crummey, John M., and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9(1):21–65.

Merritt, Rick. 2010. Interview with chuck thacker, pc pioneer.

Meyer, Bertrand. 1988a. Bidding farewell to globals. *Journal of Object-Oriented Programming (JOOP)* 1(3):73–77.

———. 1988b. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*.

Michael, Maged M. 2004. Scalable lock-free dynamic memory allocation. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI)*, 35–46.

Michael, Maged M., Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent work stealing. In *Proceedings of the 14th symposium on Principles and Practice of Parallel Programming*, 45–54.

MIPS. 2008. *MIPS64 architecture for programmers volume II: The MIPS64 instruction set*.

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

Moir, Mark, and Nir Shavit. 2004. Concurrent data structures. In *Handbook of Data Structures and Applications*, 47–14–47–30. Chapman and Hall/CRC Press.

Moore, Gordon E. 1965. Cramming more components into integrated circuits. *Circuits* 114–117.

Moshovos, Andreas, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1997. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 27th annual International Symposium on Computer Architecture (ISCA)*, 181–193.

Netzer, Robert H. B., and Barton P. Miller. 1992. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Langauges and Systems (LOPLAS)* 1(1):74–88.

Noble, James, Jan Vitek, and John Potter. 1998. Flexible alias protection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 158–185.

Olukotun, Kunle, and Lance Hammond. 2005. The future of microprocessors. *Queue* 3(7):25–29.

OpenMP. 2008. *OpenMP application program interface (version 3.0)*. OpenMP Architecture Review Board.

Owens, John D., David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1):80–113.

Papadopoulos, Dionysios P. 1998. Hood: A user-level thread library for multiprogramming multiprocessors. Master's thesis, University of Texas at Austin.

———. 1999. Hood library. `http://freshmeat.net/projects/hood/`.

Parnas, David L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12):1053–1058.

Perlis, Alan J. 1982. Epigrams on programming. *SIGPLAN Notices* 17(9):7–13.

Pierce, Benjamin C. 2002. *Types and programming languages*. MIT Press.

Preparata, F. P., and M. I. Shamos. 1985. *Computational geometry—an introduction*. Springer-Verlag.

Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22(2):416–430.

Reinders, James. 2007. *Intel threading building blocks*. O'Reilly & Associates, Inc.

Rinard, Martin C., and Monica S. Lam. 1998. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20(3):483–545.

Rudolph, Larry, Miriam Slivkin-Allalouf, and Eli Upfal. 1991. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual Symposium on Parallel Algorithms and Architectures*, 237–245.

Smith, James E., and Gurindar S. Sohi. 1995. The microarchitecture of superscalar processors. *Proceedings of the IEEE* 83(12):1609–1624.

SPARC. 1994. *The sparc architecture manual (version 9)*. Englewood Cliffs, New Jersey: PTR Prentice Hall.

Steele, Guy L. 1990. *Common LISP: The language*. 2nd ed. Digital Press.

Stroustrup, Bjarne. 1995. Why c++ isn't just an object-oriented programming language. *OOPS Messenger* 6(4):1–13.

———. 1997. *The C++ programming language*. 3rd ed. Addison-Wesley.

Sălcianu, Alexandru, and Martin Rinard. 2005. Purity of side effect analysis for java programs. In *Proceedings of the sixth international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 199–215.

Sutter, Herb. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30(3).

Sutter, Herb, and James Larus. 2005. Software and the concurrency revolution. *Queue* 3(7):54–62.

Tanenbaum, Andrew S. 2001. *Modern operating systems*. 2nd ed. Prentice Hall.

Taura, Kenjiro, Kunio Tabata, and Yonezawa Akinori. 1999. StackThreads/MP: integrating futures into calling standards. In *Proceedings of the seventh symposium on Principles and Practice of Parallel Programming (PPoPP)*, 60–71.

Tsigas, Philippas, and Yi Zhang. 2003. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Proceedings of the 11th Euromicro conference on Parallel, Distributed, and network-based Processing (PDP)*, 372–381.

Vakilian, Moshen, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. 2009. Inferring method effect summaries for nested heap regions. In *Proceedings of the international conference on Automated Software Engineering (ASE)*, 421–432.

Van Horn, Earl Cornelius, Jr. 1966. Computer design for asynchronously reproducible multiprocessing. Ph.D. thesis, Massachusetts Institute of Technology.

Vaziri, Mandana, Frank Tip, and Julian Dolby. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd symposium on Principles Of Programming Languages (POPL)*, 334–345.

Veldhuizen, Todd. 1995. Using C++ template metaprograms. *C++ Report* 7(4): 36–43.

Wulf, William, and Mary Shaw. 1973. Global variable considered harmful. *SIGPLAN Notices* 8(2):28–34.

Yeager, Kenneth C. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro* 16(2):28–40.