26

Parallel Compilation on a Tightly Coupled Multiprocessor

by Mark Thierry Vandevoorde

March 1, 1988

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Parallel Compilation on a Tightly Coupled Multiprocessor

Mark Thierry Vandevoorde

March 1, 1988

Submitted in partial fulfillment of the requirements for the degree of Master of Science, Massachusetts Institute of Technology, May 22, 1987.

Author's abstract

This thesis describes a C compiler that runs in parallel on a tightly coupled multiprocessor. The compiler, called PTCC, consists of a two-stage pipeline. The first stage performs extended lexical analysis for the second stage, which does the parsing and assembly code generation. The second stage processes units of the source program concurrently. Units as fine as a single statement are compiled in parallel.

To avoid unproductive concurrency, a new scheduling abstraction, called WorkCrew, is used in PTCC. In the WorkCrew model of computation, the client creates tasks and specifies how they can be subdivided. WorkCrews favor serial execution when parallel execution is unproductive and coarser grains of parallelism over finer ones.

Several experiments were done to measure the performance of PTCC. With 5 processors, PTCC performed up to 3.3 times better than a similar sequential compiler.

Mark Thierry Vandevoorde

Capsule review

Parallel compilation is important to computer science for two reasons. First, parallel techniques make it possible for compilers to take advantage of multiprocessor architectures to improve the real-time responsiveness of a programming environment. Second, because the structure of a compiler offers opportunities for concurrency at several distinct levels, parallel compilers provide a useful testbed for understanding the structure and performance of parallel algorithms in a more general way.

This report describes a prototype implementation of a parallel compiler and examines how compiler performance is affected by the number of parallel threads and the granularity of the subtasks. Unlike previous work in this area, which tends to restrict the parallel decomposition to the procedure level, this report measures the effect of carrying this decomposition through to individual statements.

This report also introduces a dynamic strategy called WorkCrews, which is used to control parallelism and reduce the associated overhead. This technique is quite general and is appropriate for a variety of concurrent applications.

Eric Roberts

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Compilers are among the tools most heavily used by programmers. Therefore, the speed of a compiler can have a great impact on the process of programming. One way to speed up compilation is to exploit parallelism.

Other techniques used to reduce compile time include incremental compilation and separate compilation of program modules. Incremental compilers save time by recompiling only those units of a program that depend on changes made since the last compilation. Separate compilation is incremental compilation where the units are files (program modules). Parallel compilation can make both of these techniques faster.

1.1 A Parallel C Compiler

This thesis describes the development, implementation, and measurement of an algorithm for parallel compilation on a tightly coupled multiprocessor. It also introduces a method for dynamically restricting unproductive parallelism at runtime. The parallel compiler developed, called the Parallel Titan C Compiler (PTCC), is an extension of an existing syntax-directed C compiler which generates assembly code. This research does not address the problem of designing a parallel optimizer.

Goals in developing PTCC were that it:

- use as many processors as possible
- exploit concurrency efficiently
- minimize dependencies on the source program.

Two techniques have been used to obtain concurrency in compilers running on distributed systems: pipelining stages of compilation, and processing units of the source language in parallel. PTCC exploits both forms of concurrency. It uses a two-stage pipeline in which scanning is done by the first stage while the second stage parses and generates code. The second stage runs in parallel, processing units as fine as a simple statement concurrently. Unlike code, all declarations in a block are processed serially.

PTCC differs from previous parallel compilers in several ways. First, it extends the role of the scanner to include matching delimiters for the parser. This allows the parser to advance past units of the source program quickly.

Second, PTCC uses finer grains of parallelism than have been used on distributed systems. Previous compilers have processed only procedures in parallel; PTCC processes units as small as a simple statement concurrently.

Finally, it runs on a tightly coupled multiprocessor. Since such computers have only recently become widely available, most previous parallel compilers have been built on top of distributed workstations [7, 12, 4]. Therefore, the performance possible on a tightly coupled system is not yet well understood.

1.2 Overview of Related Work

This section briefly describes other efforts related to parallel compilation. Some previous work has been purely of a design nature, and some has involved implementations of parallel compilers on distributed systems. The methods which have been studied include concurrent:

- pipelining
- processing of units of the source program

- evaluation of attribute grammars
- shift/reduce parsing.

Compilers that use the first two methods are discussed in greater detail in chapter 2.

Lipkie is perhaps the first to have suggested pipelining and concurrent processing of procedures as general techniques for parallel compilation [10]. His thesis proposes a design for a parallel compiler which uses both methods, but it does not report on any implementation.

Miller and LeBlanc wrote a pipelined compiler for Jigsaw, a subset of Pascal [12]. The compiler consists of three stages running on three computers connected by a local area network. Frankel extended a one-pass recursive-descent Pascal compiler to process procedures concurrently [7]. Whenever it encounters a procedure definition, it creates a new compiler instance to process the definition and skips to the end of the procedure. Miller's and Frankel's compilers are discussed in greater detail in chapter 2.

Seshadri, Small, and Wortman are in the process of building a parallel compiler to run on a distributed system [14]. Their design is based solely on processing units of the source program concurrently, using syntax-directed translation. Their project is more ambitious than PTCC because they are examining the possibility of processing declarations concurrently. The latter introduces what they call the "doesn't know yet" problem: the compiler may attempt to access information introduced by a declaration before it has been processed.

1.2.1 A Parallel Attribute Grammar Evaluator

Compilers based on pure attribute grammar evaluation differ from those based on syntax-directed translation like PTCC and Frankel's recursive-descent compiler. In a syntax-directed translator, pieces of the input are translated as the parser processes the input. In translators based on attribute grammars, attributes are associated with grammar symbols. Productions applied by the parser merely define functional relationships between attributes. Because the relationships between attributes must be purely functional (free of side effects), independent attributes can be computed concurrently. (See [1] for a detailed description of attribute grammars.)

Boehm and Zwaenepoel developed a compiler-generator which produces parallel evaluators for attribute grammars [4]. With this tool, they built a compiler for a subset of Pascal. The compiler, like Frankel's, processes procedures in parallel. It runs on SUN-2 workstations connected by an Ethernet. First, the input is scanned and parsed. Then, the parser divides the parse tree into approximately equal pieces – one for each attribute evaluator. Each evaluator then processes its subtree, transmitting and receiving shared attributes as necessary.

For efficiency, Boehm and Zwaenepoel use both static and dynamic attribute evaluators. Once the entire program has been parsed, a dynamic evaluator creates a graph of the dependencies between attributes. It then evaluates the attributes in an order analogous to a topological sort of the graph. A static evaluator, however, never creates a dependency graph. Instead, it evaluates the attributes in a standard order determined by the compiler-generator. Boehm and Zwaenepoel use static evaluation, which is more efficient, to evaluate all attributes local to a processing node. Dynamic evaluation, which is more powerful, is required for attributes which are shared between nodes.

The maximum speedup Boehm and Zwaenepoel observe for a 1000-line program is approximately 2.5, using five workstations. They are currently tuning their compiler and intend to experiment with statement-level decomposition.

1.2.2 Parallel Shift-Reduce Parsing

One area of research related to parallel compilation is parallel parsing. Mickunas and Schell describe a parallel algorithm for shift-reduce parsing and a method for computing the necessary tables [11]. Their work is of a design nature; they do not build a parallel parser. Their algorithm uses multiple parsers, each working on an arbitrary segment of the token stream. All parsers except the leftmost one, which starts at the beginning of the input, start in a super-initial state that accepts postfixes of the original language and may introduce conflicts in the grammar. When a parser encounters such a conflict or is unable to perform a reduction, it sends its stack to the neighbor to its left and picks up where it left off. Once it has exhausted its segment of the source, a parser merges the stacks it receives from the neighbor to its right.

Cohen and Kolodner constructed tools for estimating the performance of parallel shift-reduce parsers [5]. One tool was a simulator whose parameters included the number of processors and the relative costs of the shift, reduce, and merge operations. They found that when the three operations are equal in cost, the performance of the algorithm levels off when the number of processors used is approximately 5% of the length of the input. However, their analytical model did not account for the cost of communication between processors.

Typically, parsing represents a relatively small part of the total compilation time. Therefore, parallel shift-reduce parsing alone will not significantly reduce compilation time – other sources of concurrency are needed.

1.3 Overview of Remainder of Thesis

Chapter 2 describes two general techniques, pipelining and concurrent processing of blocks, that can be used to make compilers run in parallel. It discusses, in a general setting, issues that arise in writing a parallel compiler. These issues include the impact of the granularity of parallelism on performance and the flow of information during compilation.

Chapter 3 explains how, using the methods developed in chapter 2, I extended an existing C compiler to run in parallel. This third chapter identifies both the desirable and undesirable properties of C when applying the techniques developed in chapter 2, and it lists the changes required in TCC, the original C compiler, to support parallel compilation.

Once practical opportunities for concurrent execution are identified, a strategy to control parallel execution is needed to improve efficiency. This is the subject of chapter 4, which describes WorkCrews. WorkCrew is a scheduling abstraction that improves the efficiency of parallel programs by avoiding unproductive concurrency. In the WorkCrew model of computation, a set of workers (processors) cooperate to solve a problem by dividing it into individual tasks. Chapter 4 also explains how WorkCrews are used in PTCC.

Chapter 5 contains a performance analysis of PTCC. The primary experiment was to measure the elapsed time and CPU time required by PTCC to compile a set of files.

Two compiler parameters were varied for each file: the granularity of parallelism used, and the number of workers used. Chapter 5 also quantifies the effects of bus contention, insufficient parallelism, and concurrency overhead on the performance of PTCC.

Chapter 6 sketches how error recovery might be performed in a parallel compiler. It also explains how the techniques used in PTCC can be applied to compilers of multipass languages.

Chapter 7 summarizes the conclusions of this research. The major conclusion is that parallel compilation can reduce compilation time significantly. Furthermore, statement-level compilation, though typically unnecessary given only five processors, sometimes offers significant improvement over procedure-level compilation.

1.4 The Firefly

PTCC runs on the Firefly, a tightly coupled multiprocessor developed at Digital Equipment Corporation's System Research Center [15]. A Firefly may have from one to seven MicroVax II processors sharing its single bus.[1] All memory is global, and each processor has a 16-kilobyte cache. Because the hardware maintains the caches, cache management is not a software issue.

Processors may communicate by writing and reading the shared memory. Binary semaphores, which are implemented with the test-and-set instructions of the Vax, are used to control access to shared memory. Furthermore, all accesses to aligned memory words are atomic.

The caches of the Firefly are used to reduce bus traffic. The cycle time of the memory and bus is twice the maximum memory access rate of any single processor. Therefore, up to two processors may access memory at the maximum rate without bus contention provided that memory references are perfectly interleaved. Every cache miss requires a bus cycle to read the main memory. Furthermore, every write to locations cached by other processors requires a bus cycle to maintain cache consistency.

[1] Only Fireflies with up to five processors were available when I ran my parallel compiler.

The primary programming environment for the Firefly is based on Modula-2+, which is Modula-2 with extensions for garbage collection, exception handling, and concurrency [16, 13]. For this work, the principal new feature is, of course, concurrency.

The primary concurrency abstraction in Modula-2+ is the `Thread` module. A *thread* is a process in a shared address space. Threads are manipulated using a fork/join semantics.

In Modula-2+, `Thread.Fork` takes two arguments, the first of which is a procedure. `Fork` creates and returns a new thread which invokes the procedure on the second argument. `Thread.Join` takes a thread as an argument. It suspends the caller until the thread has completed. `Join` passes the return value of the procedure passed to `Fork` to the caller.

Although Modula-2+ does not impose a hierarchy on threads, it is often useful to impose an abstract hierarchy. If thread A forks thread B, then A is the *parent* of B, and B is the *child* of A.

Appendix A contains *Thread.def*, the definition module for threads. *Thread.def* also defines a semaphore abstraction, `Thread.Mutex`, used to synchronize threads.

Chapter 2

Algorithms for Parallel Compilation

This chapter describes two general techniques that can be used to make compilers run in parallel: pipelining, and processing source language blocks concurrently using syntax-directed translation. Both of these have been studied on distributed computers. Miller and LeBlanc built a pipelined compiler [12], and Frankel added concurrency to a recursive-descent compiler [7]. I use both techniques in PTCC.

2.1 Pipelining

2.1.1 General Pipelining

Pipelining is a common technique used to exploit concurrency in tasks that can be divided into a series of stages. The stages are cascaded so that the output of one stage is the input for the next. Some form of buffering is needed between each pair of stages to facilitate the transfer of information from one stage to the next. If each stage runs concurrently, the maximum speedup is equal to the number of stages.

Pipelining has several limitations. First, if there are fewer stages than processors, a pipeline will not be able to utilize a multiprocessor fully. Second, the stages may be unbalanced. A pipeline can be no faster than its slowest stage, so if 50% of the CPU time is spent in one stage, then pipelining can offer a speedup of two at best. Much of the effort to implement an efficient pipeline may go to balancing the different stages. Finally, pipelining can introduce significant overhead. Each stage represents an additional pass over the input, and buffering is required between stages.

2.1.2 Pipelining Compilation

Information Flow in Compilers

The first step in designing a pipeline is to analyze the flow of information in the computation of the task. Figure 2.1 depicts the flow of information in many compilers. Here, the compiler uses several passes. First, a scanner converts the input stream of characters into a stream of tokens. Second, a parser converts the token stream into a parse tree. Third, a symbol table is built by walking over the parse tree. Fourth, intermediate code is generated from the parse tree and the symbol table. Finally, an optimizer improves the intermediate code and constructs an executable file. Often, the optimizer is subdivided into multiple stages. Some languages also provide a preprocessor; this can be added either before or after the scanner in figure 2.1. Although each pass is conceptually distinct, multiple passes are often merged for efficiency reasons.

A Compilation Pipeline

A simple way to pipeline a compiler is to have a thread for each pass of the compilation. For example, a scanner thread reads the characters of the source file and converts them into tokens for the parser thread. The parser thread converts the tokens into a parse tree for the code generator, and so on.

Each pair of adjacent stages has a producer/consumer relationship. The type of data passed between stages varies, but it has some linear form. The scanner might create a stream of tokens, which the parser would transform into a stream of semantic actions. Because stages run concurrently, mutexes must be used to coordinate access to the data stream. To reduce synchronization costs, multiple data units can be transferred each time the mutex is acquired.

It is important to note that pipelining is probably not a feasible strategy for the front

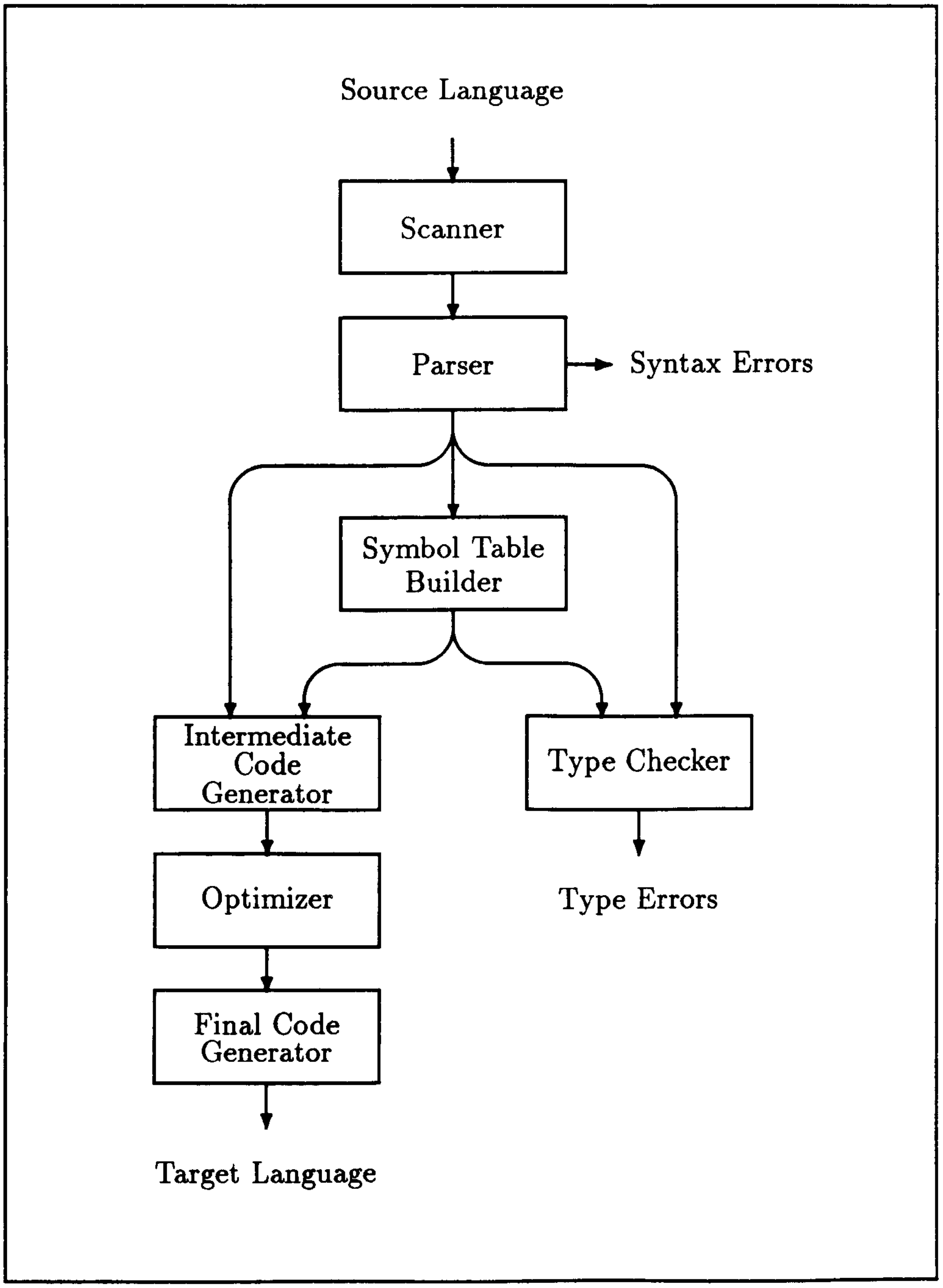

Figure 2.1: Information flow in compilers.

ends of compilers of multipass (as opposed to one-pass) languages. For example, any type-checked language which allows an identifier to be used before it is declared requires at least two passes: one to build the symbol table, and a second to check for type errors. In general, the second pass cannot begin until the first pass is complete, since a declaration at the end of the source file may be needed to type-check code at the beginning of the source file. If a compiler for such a language were pipelined, then the opportunities for concurrency would depend on the ordering of declarations and references.

2.1.3 A Distributed Pipelined Compiler

Miller and LeBlanc wrote a pipelined compiler for Jigsaw, a subset of Pascal [12]. The compiler consisted of three stages: a lexical analyzer, a syntactic analyzer, and a semantic analyzer. The lexical analyzer tokenized the input, passing information to the two other stages. The syntactic analyzer converted a stream of tokens into a stream of semantic actions, which the semantic analyzer combined with information from the lexical analyzer to generate code.

Miller ran the compiler on three distributed computers (two Prime 550's and one slightly slower Prime 400) connected by a local area network. Each stage was put on one computer; the semantic analyzer ran on the 400. The syntactic analyzer was the slowest stage. Miller also ran a sequential version of the compiler, in which communication was done via procedure invocation on a single computer. For programs longer than 100 lines, Miller observed a speedup factor between 2 and 2.5. As one might expect, Miller found that buffering of information between stages is critical to performance in a distributed compiler. Without buffering, the speedup factor was at best 1.4.

2.2 Processing Code Concurrently

When performing parallel compilation on a large-scale multiprocessor, pipelining is not enough. Additional parallelism may be possible by exploiting the structure of the input to a program. When a problem can be decomposed into several independent parts, a divide-and-conquer approach is possible. Divide-and-conquer creates multiple threads to solve independent subproblems and then merges the results. If a problem is tree-structured, divide-and-conquer may be used recursively.

This section explores when and how a divide-and-conquer approach can be used to compile programs. It focuses on the tree-structure of programs. Although the technique described uses a top-down recursive-descent parser, the same basic strategy could be used for deterministic bottom-up parsers.

2.2.1 Grammars of Programming Languages

Figure 2.2 is an abstract grammar for typical programming languages. The grammar defines a program as a sequence of declaration and code units. The undefined nonterminals **decl** and **stmt** correspond to the normal intuitive concepts of *declaration* and *simple statement.* A *simple statement* is any statement which does not itself contain statements.

In this example, I use **decl** to represent any construct which introduces information required to compile other constructs in the source file. The **stmt** construct, on the other hand, is self-contained: it provides only information required to compile itself, although it typically also uses information introduced by **decl**s. The grammar is intended to be as general as possible; therefore, it allows **decl**s and **stmt**s to be intermixed freely.

Most programming languages also provide constructs for compound statements and nested blocks. In the grammar of figure 2.2, the nonterminal **code** may designate either structure. Both constructs allow a sequence of **stmt**s to be grouped into a single unit; the difference is that nested blocks may contain **decl**s as well as **stmt**s. Production 5 of the grammar represents nested blocks. Since compound statements can be viewed as nested blocks containing no **decl**s, I did not introduce a production for them.

2.2.2 Constraints on the Flow of Information

Through their scoping rules, lexically scoped languages constrain the flow of information introduced by declarations. Scoping requires that all references to a declaration be from **code** units in the same scope as the declaration. In terms of the grammar

```
program ::= decl program              (1)
          | code program              (2)
          | <empty>                   (3)

code ::= stmt                         (4)
       | begin program end            (5)
```

Figure 2.2: Grammar for a generic programming language.

of figure 2.2, the scope of a declaration is its smallest enclosing **code** unit. Lexical scoping decouples sibling **code** units from one another and makes the parent/child coupling unidirectional. (The parent knows nothing of the child, but the child knows all about its parent and ancestors.)

Some languages impose another constraint: declarations must appear in the source file before they are used. Whereas the constraints of lexical scoping are usually considered desirable because they enhance modularity, requiring declaration before use is sometimes a hindrance to programmers.[1] The purpose of this constraint is to allow compilers for the language to run using only one pass, making the compilers faster.

For languages which are both one-pass and lexically scoped, such as C and Pascal, these constraints can be restated as the following properties:

1. All the information required to process the contents of a **code** unit appears *before* the **code** unit itself.
2. The contents of a **code** unit are not needed to process the contents of outer **code** units or **decl** units.

2.2.3 The Parallel Algorithm

Together, the two constraints on information flow make the parallel recursive-descent compilation strategy described in figure 2.3 possible. The top-level procedure,

[1] For example, FORWARD procedure declarations in Pascal are a nuisance.

Program, compiles a sequence of **decl** and **code** units. However, whenever a **code** unit is encountered, it forks a thread to compile the **code** unit and skips to the next unit. The parent thread must then buffer its output until the child completes. The child thread is inserted into the set CompilerThreads. All threads in this set must be joined before the compilation terminates.

If the source language has the properties mentioned earlier, then both the parent and child threads can proceed independently. Property 1 ensures that the child will not have to wait for further information from the parent. All of the information required to compile the **code** unit must have appeared in **decl** units prior to the the **code** unit, and these **decl** units were processed before the child was forked. Similarly, property 2 ensures that the parent will not have to wait for information from the child before continuing. **Decl** units nested in a **code** unit do not affect outer units.

The algorithm outlined in figure 2.3 may fork threads recursively since procedures Code and Program are mutally recursive. It is possible that nodes at each level of the parse tree may be processed concurrently, but note that **decl** units of a scope are always processed serially. Unlike pipelining, which offers a fixed performance improvement, processing **code** units in parallel is limited only by the structure of the source file and the number of available processors. Furthermore, if the algorithm of figure 2.3 is implemented using the technique described in chapter 4, an upper bound on the number of threads used can be specified (and altered) by the user or the operating system at runtime.

Although the **decl** units of each scope are always processed serially, **decl** units at different levels in the parser tree may be processed concurrently. Therefore, care must be taken to ensure that programs with use-before-declaration errors are still considered illegal. Using the algorithm described in figure 2.3, there is a race between the processing of a variable declaration and uses of the variable. By adding position stamps to each entry of the symbol table, it is possible to determine if a variable is used before it is declared.

```
VAR CompilerThreads:  SetOfThreads.T; (* the set of all compiler threads *)

PROCEDURE Program(VAR ts:  TokenStream.T);
  (* Effect: Compiles a program and advances ts past the tokens of the program. *)
  BEGIN
    WHILE NOT TokenStream.Empty(ts) DO
      IF the next token denotes the beginning of a declaration
         THEN Decl(ts);
         ELSE SetOfThreads.Insert(CompilerThreads, Thread.Fork(Code,ts));
              SkipCode(ts);
      END;
    END;
  END Program;

PROCEDURE Decl(VAR ts:  TokenStream.T);
  (* Modifies: ts.                                                    *)
  (* Effect: Compiles a declaration and advances ts past the tokens of *)
  (*         the declaration.                                          *)
  END Decl;

PROCEDURE Code(ts:  TokenStream.T);
  (* Effect: Compiles a code unit. *)
  BEGIN
    IF the next token is a BEGIN
       THEN TokenStream.Get(ts); (* Consume the BEGIN *)
            Program(ts);
            verify that the next token is an END and consume it
       ELSE Stmt(ts);
    END;
  END Code;

PROCEDURE Stmt(ts:  TokenStream.T);
  (* Effect: Compiles a statement. *)

PROCEDURE SkipCode(VAR ts:  TokenStream.T);
  (* Modifies: ts.                                                    *)
  (* Effect: Advances ts past the tokens for the next code unit in ts . *)

(* In the TokenStream Module: *)

PROCEDURE Peek(ts:  T): Token.T;
  (* Effect: Returns the first token in ts . *)

PROCEDURE Get(VAR ts:  T): Token.T;
  (* Modifies: ts.                                                          *)
  (* Effect: Returns the first token in ts and advances ts to the next token. *)
```

Figure 2.3: Parallel recursive-descent compiler

2.2.4 Synthesized Attributes

The second constraint of section 2.2.2 prohibits a **code** unit from using information gathered by other **code** units. Thus, **code** units may not have synthesized attributes.

Often, compilers associate an intermediate-code attribute with each **code** unit; the attribute's value is the compiled code. This attribute can be eliminated by outputting the compiled code directly. However, eliminating the use of synthesized attributes is not always possible.

For example, suppose the target language provides a bounded number of registers. Once the registers have been exhausted, the compiler must use stack temporaries. To determine how many stack temporaries are required, the compiler might associate a synthesized attribute *NTemps* with each **code** unit. *NTemps* for a compound **code** unit is computed by taking the maximum of the *NTemps* of its nested **code** units.

Figure 2.4 shows how the algorithm might be revised to allow the synthesized attribute *NTemps*. The primary change is that `Program` delays its return until all of the threads it has forked terminate. (This is implemented using `Thread.Join`, which yields the return value of the procedure invoked when the thread was forked.) Now, whenever `Program` or `Code` return, any threads forked to process **code** units must have terminated. Therefore, the synthesized attributes of **code** units are available when `Code` returns.

Because the use of synthesized attributes violates the second constraint of section 2.2.2, the assertion that the parent will never have to wait for information computed by the child is no longer true. The parent may block whenever it uses a synthesized attribute of the child.

2.2.5 Advancing Past Code Units Efficiently

Recall that whenever a thread is forked to process a **code** unit, the forker must then skip to the next unit. In figure 2.3, `Program` calls `SkipCode` after each call to `Fork`. Unless `SkipCode` is significantly faster than `Code`, the parallel strategy will not produce any gains in performance. In a simple, general approach, the skipped **code** unit must be parsed. However, by exploiting the structure of the source language's

```
PROCEDURE Program(VAR ts:  TokenStream):  INTEGER;
  (* Effect: Compiles a program and advances TS past the tokens of *)
  (*         the program. Returns the number of temporaries used.   *)
  VAR
    ntemps:  INTEGER;
    thread:  Thread.T;
    threads:  SetOfThreads.T;
  BEGIN
    ntemps := 0;
    threads := SetOfThreads.Create();
    WHILE NOT TokenStream.Empty(ts) DO
      IF the next token denotes the beginning of a declaration
         THEN Decl(ts);
         ELSE
           SetOfThreads.Insert(threads, Thread.Fork(Code,ts));
           SkipCode(ts);
      END;
    END;
    WHILE NOT SetOfThreads.Empty(threads) DO
      thread := SetOfThreads.DeleteAnyMember(threads);
      ntemps := MAX(ntemps, Thread.Join(thread));
    END;
    RETURN ntemps;
  END Program;

PROCEDURE Code(ts:  TokenStream):  INTEGER;
  (* Effect: Compiles a code unit; returns the number of temporaries used. *)
  VAR
    ntemps:  INTEGER;
  BEGIN
    IF the next token is a BEGIN
       THEN TokenStream.Get(ts); (Consume the BEGIN)
            ntemps := Program(ts);
            verify that the next token is an END and consume it
       ELSE RETURN Stmt(ts);
    END;
    RETURN ntemps;
  END Code;

PROCEDURE Stmt(ts:  TokenStream):  INTEGER;
  (* Effect: Compiles a statement; returns the number of temporaries used. *)
  END Stmt;
```

Figure 2.4: Example using synthesized attributes.

syntax, complete parsing can be avoided.[2]

Consider again the grammar in figure 2.2. Note that nested blocks are always surrounded by **begin** and **end**. By extending the scanner to match each **begin/end** pair, SkipCode can be made extremely fast. Note that such a scanner is an elementary parser: it will have to keep a stack of pointers to unmatched **begin** tokens. Each **begin** token will have a *match* field which will point to the matching **end** in the token stream. Whenever an **end** is encountered, the *match* field of the **begin** on the top of the stack is set. Because Code does not need to read the *match* field of **begin** tokens, the processing of a nested block is *not* delayed until the entire block is scanned.

The strategy above requires that syntactic structures of the source language be delimited by some set of reserved words, and the delimiting reserved words must not be used elsewhere.[3] The set of delimiting reserved words is partitioned into two subsets, *Start* and *Stop*, which contain the tokens that delimit the beginning and end, respectively, of syntactic structures. It is not necessary for each syntactic structure to have its own pair of delimiters. For example, *Start* might contain **if**, **while**, and **repeat**, with *Stop* containing only **end**. The scanner needs to distinguish only the sets *Start* and *Stop*. It can ignore distinctions between elements of the same set and let the parser handle inappropriately paired delimiters.

Fortunately, large syntactic structures are usually delimited by a pair of reserved words. However, smaller syntactic structures such as simple statements may not be delimited as required. If the language requires a statement terminator then the parser can skip past a statement by advancing to the next terminator. If statement separators are used instead, then the parser can skip past a statement by advancing to the next separator or member of *Stop*. In both cases, the symbol used to delimit statements must not occur within them. If neither a terminator nor a separator is required, then the skipped statement must be parsed. In this case, parallel compilation of simple statements may not be practical.

Seshadri, Small, and Wortman independently developed a similar algorithm for advancing past units of the source program [14].

[2] If the compiler is integrated with a syntax-oriented editor, an alternative is possible: the editor can produce a parse tree for the compiler. In this case, skipping past code units is trivial.

[3] Languages which have keywords rather than reserved words are unsuitable for this strategy because keywords may be used as identifiers in certain contexts.

Frankel extended a one-pass recursive-descent Pascal compiler to process procedures concurrently using the algorithm of figure 2.3 [7]. Whenever it encounters a procedure definition, the compiler creates a new instance to process the definition and skips to the end of the procedure by matching delimiters. Error recovery is restricted to deleting tokens until either an expected token is encountered or the end of file is reached.

Frankel's compiler ran on Xerox Alto workstations connected by an Ethernet. Processors communicate via a file server, and the `Fork` operation must manipulate idle machines through the network.

Frankel observed a speedup of 3.74 when using six workstations to compile a 6000-line program (2.07 for three workstations). However, he partly discounts these numbers because his compiler, which was built on multiple levels of interpretation, was CPU-bound. The network was not a bottleneck. When communication costs are negligible, the addition of processors is expected to increase performance significantly.

Chapter 3

A Parallel C Compiler

This chapter describes the design and implementation of PTCC, a C compiler which exploits concurrency using the methods decribed in chapter 2. Rather than writing a compiler from scratch, I chose to add parallelism to an existing compiler, the Titan C Compiler (TCC).[1]

3.1 The Titan C Compiler (TCC)

The Titan C Compiler is a recursive-descent compiler written in Modula-2. It compiles preprocessed C, which contains no comments or macros, into the Mahler intermediate language. Mahler is a source-language independent intermediate language designed for the Titan instruction set. TCC makes no attempt to recover from syntax errors, and I did not add such facilities to PTCC. Although not extensively used, TCC has compiled large programs, including the UNIX operating system.

Before I could begin to experiment with TCC, I had to translate TCC from Modula-2 to Modula-2+. Translation involved changes in syntax, data types, and system calls, which were modified to reflect the conversion from Ultrix to Topaz, the operating system of the Firefly.

[1] The Titan is a RISC computer developed for research at Digital's Western Research Laboratory.

3.2 Relevant Specifics of C

3.2.1 Decl and Code Units

In C, any variable or type declaration is a **decl** unit: static initializers, **typedefs**, **structs**, **unions**, etc. are all **decl** units. Any procedure body or statement is a **code** unit.

A procedure definition is a **decl** unit followed by a **code** unit: the procedure header is a declaration, and its body is code.

3.2.2 Exploiting C's Syntax

C's syntax has most of the characteristics, listed in section 2.2.5, that allow code blocks to be skipped efficiently. All compound statements, procedure bodies, and nested blocks are delimited by braces, which do not appear elsewhere in **code** units. C's simple statements are all terminated by semicolons, which do not appear elsewhere in simple statements.

C's structured statements, such as **if**, **while**, and **for**, are not bracketed by any delimiters. Each structured statement begins with a unique reserved word, but none is terminated by a delimiter. Therefore, structured statements must be partially parsed. Several of the structured statements (**if**, **for**, **while**, and **switch**) contain an expression enclosed in parentheses, which always occur in matched pairs. By having the scanner match parentheses as well as braces, it is possible to skip over the structured statements more efficiently.

3.2.3 Violations of Block Structure in C

GoTo

Perhaps the most prominent violation of block structure in C is the **goto** statement. Forward **goto**'s violate the first constraint of section 2.2.2. Furthermore, C allows the target of a **goto** to be any statement in the program; this violates the second constraint.

If the target language has global symbolic labels, then one-pass parallel compilation is possible: the compiler simply reuses the label names of the source program. PTCC uses this strategy. If the target language does not provide symbolic labels, then compilation requires two passes.

Switch Statement

C's **switch** statement violates block structure because the case arms are equivalent to labels that may be inserted before any statement inside the switch body. For example, the program in figure 3.1 is legal. Note that each case header is a **decl** rather than a **code** unit: its location and the value of its expression are needed to compute a dispatch table.

One way to handle switch statements would be to treat the switch bodies as **decl** units as defined in chapter 2. This has the serious disadvantage of serializing the compilation of the switch body, which may be quite large. A better strategy is to treat case arms as synthesized attributes of the switch body and to do parallel compilation as described in section 2.2.4. This is the approach used in figure 3.2. Each child stores any case arms it encounters in a shared data structure, embedded in **cs**, which is allocated and deallocated using `EnterSwitch` and `ExitSwitch`. When procedure `Statement` returns, all the information needed to generate a dispatch table is available in this data structure.

Extern

C's **extern** construct violates block structure because an **extern** declaration in a nested block may be referenced by *any* code after the **extern**. Using the algorithm of figure 2.3, there is a race between the processing of such a nested declaration and its uses outside the nested block. Since eliminating this race would eliminate all parallelism, PTCC does not promise to compile such programs properly unless it is run in serial mode. In parallel mode, references to **extern** declarations outside the scope of the declaration may be flagged as undefined variables. Programs which rely on this behavior of **extern** can be easily rewritten to avoid the problem.

```
switch (n) case 1: {
    case 2: printf("2\n"); break;
    if (foobar()) {
        case 3: n = 10;
    }
    default: n = n + 1;
}
```

Figure 3.1: C's switch statement violates block structure.

```
PROCEDURE SwitchStatement(cs:  CompilerState.T)
  (* Effect: Compiles a switch statement. *)
  VAR
    expr:  Expression.T;
    ts:  TokenStream.T;
  BEGIN
    ts := CompilerState.GetTokenStream(cs);
    ASSERT(Token.GetKind(TokenStream.Get(ts)) = TOKEN_SWITCH);
    ASSERT(Token.GetKind(TokenStream.Get(ts)) = TOKEN_LEFT_PAREN);
    expr := Expression(cs);
    ASSERT(Token.GetKind(TokenStream.Get(ts)) = TOKEN_RIGHT_PAREN);
    CompilerState.EnterSwitch(cs);

    (* Call Statement to compile the body of the switch, which is  *)
    (* usually a compound statement.                               *)
    (* Statement may execute in parallel, but when it returns, any *)
    (* threads which it may have forked will have terminated.      *)

    Statement(cs);
    GenerateDispatchTable(cs, expr);
    CompilerState.ExitSwitch(cs);
  END SwitchStatement;
```

Figure 3.2: Handling C's switch statement.

3.3 The Mahler Intermediate Language

Mahler has several features which facilitate code generation [17]:

- It supports top-level (unnested) procedures with an optional return value. Both direct and indirect recursion are allowed.
- It assumes responsibility for managing the runtime stack. Rather than generating code to do address arithmetic, the front end of the compiler uses Mahler variables, and the Mahler Compiler handles address computations. Mahler variables may be used before they are declared.
- It allows symbolic labels in conditional and unconditional branch instructions. All labels and variable names must be unique.
- It provides an essentially unlimited number of temporaries (anonymous variables) and assumes responsiblity for register allocation.

There are two types of variables in Mahler: named variables and anonymous variables. Anonymous variables are used as expression temporaries. Their values, unlike those of named variables, are not preserved across labeled instructions or transfers of control. Anonymous variables need not be declared.

Named variables may be declared to be local or static. A local variable is stored in the activation record of the current procedure; a static variable is stored in a common area. Mahler allows declaration before use. A procedure may reference the local variables of another procedure; this is used to reference free, lexically scoped variables.

3.4 A Parallel Version of TCC (PTCC)

3.4.1 Basic Design

PTCC consists of a two-stage pipeline as shown in figure 3.3. The first stage is the scanner; the second stage parses the token stream, type-checks the program, and generates Mahler intermediate code. The bulk of concurrency is obtained in the second stage by processing blocks of code in parallel.

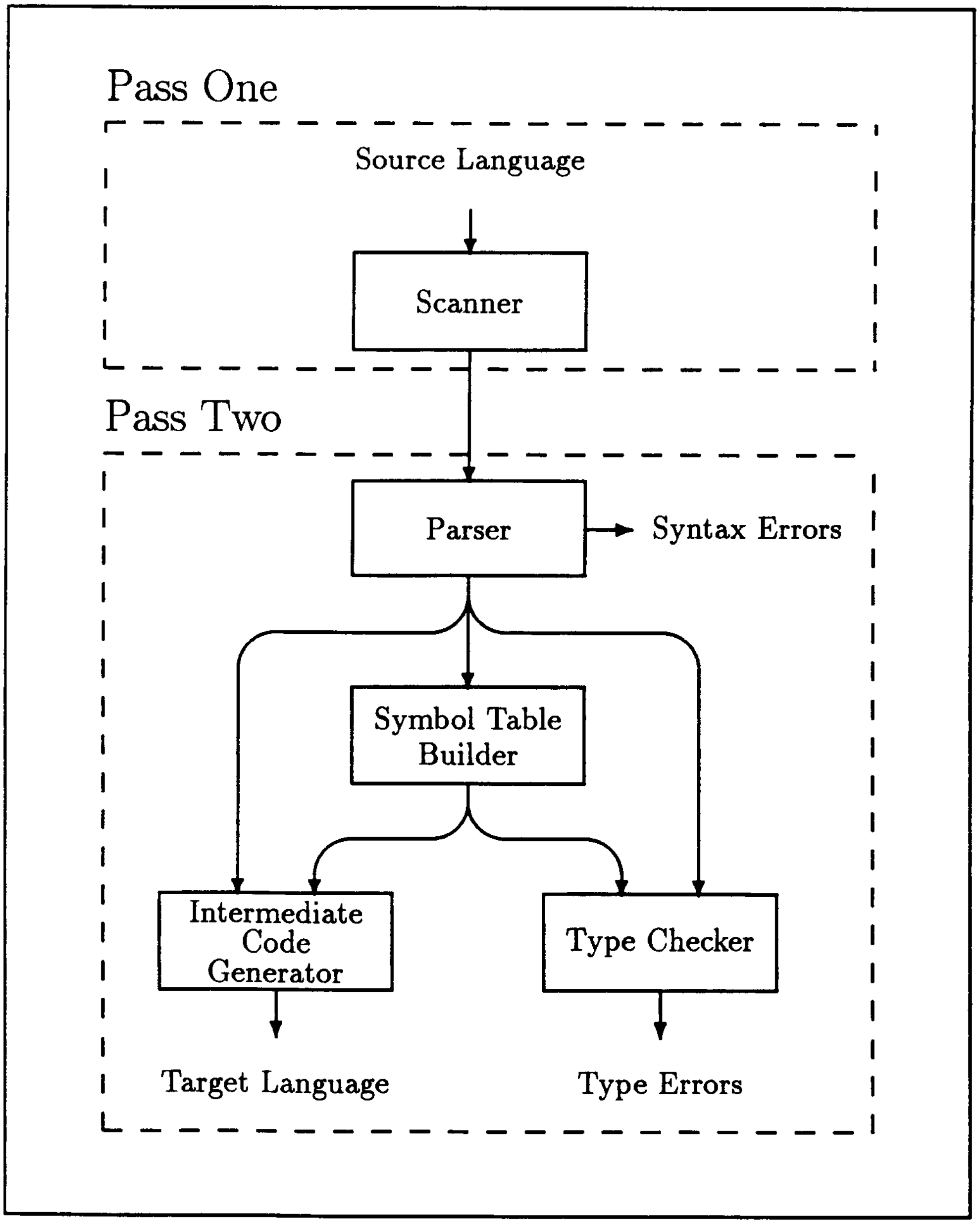

Figure 3.3: Structure of PTCC.

The pipelined scanner serves two purposes. Since scanning represents 10% to 20% of compilation time, pipelined scanning offers modest performance gains. However, the primary reason for making the scanner an independent stage is to tokenize the input quickly so that the parallel parser will not be delayed by the scanner.

I did not pipeline other stages for several reasons. Pipelining, though conceptually simple and relatively easy to implement, works best on a predetermined number of processors, and I wanted an algorithm which would scale to the number of processors available. Therefore, pipelining was not suitable as the primary source of concurrency. Once PTCC processed code blocks in parallel, more extensive pipelining seemed unnecessary for the 5-processor Firefly. In fact, the additional overhead of pipelining might have outweighed any real increases in parallelism. Moreover, PTCC is a recursive-descent compiler whose static semantics and code-generation are intertwined with the parser; separating these stages would have been difficult.

PTCC generates the same code as TCC, except that labels may be renamed.

3.4.2 Pipelining the Scanner

Pipelining the scanner with the rest of the compiler was straightforward. In TCC, the scanner is driven by the parser through a procedure **GetNextToken**. Therefore, I modified the scanner to generate the token stream independently and altered **GetNextToken** to merely return a token in the stream.

Access to the token stream by the second stage of the pipeline is restricted by valve tokens. Each valve token contains a mutex which the second stage must acquire and release before it proceeds to the next token. The token stream always begins with a valve token whose mutex is held by the scanner. Once it has added some number of tokens to the stream, the scanner inserts a new valve token and opens the previous valve by releasing its mutex.

Because the scanner is faster than the second stage in the pipeline, PTCC was not significantly delayed by the valves in practice.

3.4.3 Processing Code Blocks Concurrently

Processing code blocks in parallel required significantly more effort than pipelining the scanner. Several changes were necessary.

Eliminating Global Variables

TCC uses many global variables to store the state of the compiler. To allow multiple concurrent instances of compilers, these global variables had to be eliminated. I encapsulated them into CompilerState objects and mechanically substituted CompilerState operations for references to the global variables.

Scanner Extensions

Matching braces and parentheses in PTCC is straightforward. I added a stack to store references to each unclosed brace or parenthesis. Whenever a right brace or parenthesis is detected, the *match* field of the token at the top of the stack is set. There is actually one complication: *match* fields must never cross an unopened valve token since this would circumvent access control to the token stream. Therefore, the scanner may selectively delay initializing match fields.

Because the second stage may try to reference the *match* field before the scanner has reached the corresponding close brace or parenthesis, I implemented the *match* field of left brace/parenthesis tokens as write-once variables [2].[2] As their name implies, write-once variables may be written only once, although they may be read any number of times. Attempts to read uninitialized write-once variables are delayed until the variable is initialized.

In Modula-2+, I implemented write-once variables as a tuple <mutex, variable>. Readers of the write-once variable must first acquire and release the mutex, which is marked unavailable at creation time and then marked available by the single writer. Therefore, all reads block until the variable has been written.

[2] Write-once variables were developed by Arvind in the functional language Id designed for dataflow computers. In Id, they are used for the elements of I-structures, which are similar to arrays [2].

Because reads and writes of aligned words are atomic on the Firefly, an optimization is possible. If the write-once variable is word-aligned and is created with a value outside the legal range (e.g., NIL for pointers), then readers may first read the variable and check that the value is within the range. If so, then no further action is necessary. Otherwise, the reader must block until the variable is initialized by acquiring and releasing the mutex. This optimization avoids serializing reads once the variable has been written.

Restructuring the Symbol Table

TCC's symbol table is a tuple <*hashtable*, *scopestack*>. The *hashtable* maps identifiers to stacks of values; the top of this stack contains the identifier's value for the smallest enclosing scope in which the identifier is defined. *Scopestack* is used to maintain a list of all the local identifiers in each accessible scope; it is pushed on scope entry and popped on scope exit. When an identifier is declared, a new value is pushed onto its stack in *hashtable* and its name is added to the top of *scopestack*. On scope exit, the stacks in *hashtable* for each of the identifiers listed in the top of *scopestack* are popped.

The symbol table structure described above is not suitable for parallel compilation of code blocks. It is not possible for concurrent instances of the compiler to share common parts of the symbol table such as the global scope. Copying the symbol table each time a new instance of a parallel compiler is created would be prohibitively expensive. Therefore, I redesigned the symbol table in PTCC. The new symbol table is a tree of hash tables. Each hash table corresponds to a single scope, with the global scope at the root of the tree. The tree structure makes sharing possible; unfortunately, it increases the time for a lookup operation from a constant to a linear function of the relative depth of an identifier.

In order to detect all use-before-declaration errors, positionstamps were added to each symbol table entry. This was a small change.

Label Generator

TCC has a global label generator. In PTCC, I added a mutex to control access to the label generator, which is shared by all concurrent compiler instances. Because the order in which labels are allocated may vary each time PTCC is run, label names vary in the output code from different runs using the same source file.

Buffering Data: SplitWriters

As a one-pass compiler, TCC outputs Mahler code and error messages continuously using the standard Modula-2+ I/O facilities. Whenever a new compilation task is created, a new pair of output streams is needed (one for code, one for errors). The final output of PTCC must be the concatenation of the outputs of each compiler instance.

In Modula-2+, output is performed using a generic type Writer. A Writer is an output character stream to some target, e.g., a file or a terminal. Writers are implemented using a subclass for each type of target.

I defined a new subclass of Writer, SplitWriter, which allows parallel programs to divide a single Writer into a series of ordered Writers with a common target. SplitWriter introduces the operation `Split`, which takes a SplitWriter as an argument and returns a new SplitWriter. The semantics of SplitWriter are that everything written to the new SplitWriter will be written to the original SplitWriter once the latter is closed.

SplitWriters with a common target are logically chained in lists; the `Split` operation inserts a new SplitWriter just after its argument. The output stream of a list of SplitWriters is the concatenation of the each of the individual SplitWriter streams.

The first unclosed SplitWriter in a SplitWriter list is treated specially. It is called the *active* SplitWriter. Anything written to it is immediately written to the target. Anything written to an inactive SplitWriter is buffered by the SplitWriter abstraction. When the active SplitWriter is closed, the next SplitWriter in the list becomes active, and its buffer is flushed to the target. Therefore, output to the target is continuous as long as SplitWriters are closed as early as possible.

It is an error to try to `Split` a closed SplitWriter since output of SplitWriters later in the list may have already been flushed to the target.

Regardless of whether a SplitWriter channels its writes directly to the target writer or holds them in a temporary buffer, using SplitWriters adds a second level of buffering. (Characters are copied exactly twice.) This second level is introduced by the generic Writer abstraction.[3]

Given SplitWriters, it is easy to handle buffering in PTCC. Each time a new compiler instance is created, the output Writers are `Split`. Whenever a compiler instance completes its task, it closes its output Writers.

Putting It All Together

Once all of the pieces above were completed, it would have been possible to do parallel compilation of code blocks using the algorithm presented in chapter 2. All that is missing are the calls to **Fork** and some procedures to replicate CompilerStates when **Fork** is called. This would have led to a correct parallel compiler, but not a very efficient one because the number of threads would greatly exceed the number of processors. In the next chapter, I present a method which restricts parallelism to improve efficiency and show how it was used in PTCC.

[3] A better implementation would include SplitWriter functionality in the generic Writer abstraction so that the second level of buffering would be eliminated for the active SplitWriter.

Chapter 4

Controlling Parallelism

Often, the first step in writing a parallel program is to find all possible sources of parallelism. The second is to restrict the use of parallelism to make the most effective use of available resources. For problems with ample parallelism, the second step can be the more difficult one.

4.1 Rampant Forking Is Inefficient

On the Firefly, there is no performance advantage in having more runnable threads than available processors. Instead, this is a performance liability due to increased scheduler overhead. Furthermore, even if scheduling were free, each thread represents a division of labor. Typically, the division has a cost. For example, consider a parallel game-playing program based on a tree-search with alpha-beta pruning. Concurrent evaluation of the tree may result in less pruning because less information is shared between the parallel evaluators. Excess runnable threads, therefore, represent a liability.

In PTCC, dividing the task of compilation involves duplicating CompilerStates, creating new SplitWriters, and skipping past units of the input. These operations delay PTCC and increase its memory demands. Again, excess runnable threads represent unproductive overhead.

In chapter 2, I presented an algorithm for parallel compilation which forked a thread for each nondeclaration node in the parse tree. This uses an unproductively large number of threads. For maximum efficiency, it would be best to divide the task of compilation once, at the start, into equal parts for each processor, thereby minimizing overhead while maximizing concurrency. Such a division is impossible on the first pass since the size of the task is unknown. In general, the problem is finding an efficient way to divide a problem of unknown size and structure.

4.2 Using Fewer Threads

4.2.1 Restricting the Level of Parallelism

For recursively divisible problems, one simple way to reduce the number of threads would be to fork threads only for each top-level branch. For example, in a compiler such a strategy is analogous to restricting concurrency to the procedure level. The disadvantage with this approach is that the numbers of branches may differ from the number of processors. Moreover, even if the number of processors and branches match, the branches are not necessarily equal in size, so the multiprocessor may be underutilized towards the end of the computation.

In general, it is best to allow (but not necessarily exploit) as much concurrency as a problem will permit so as to minimize the chance of having idle processors. Eliminating opportunities for concurrency has the effect of reducing the class of input trees for which a parallel program will make full use of a multiprocessor. Furthermore, if the number of processors is increased, the program is less likely to be able to use them.

4.2.2 Workers and Task Lists

An alternative method is to use a constant number of worker threads equal to the number of processors and to maintain a list of unfinished, nonblocking tasks. This approach, which is equivalent to a nonpreemptive scheduler, is implemented by replacing the call to `Thread.Fork` in figure 2.3 with a call to `TaskList.AddTask`. One advantage of this worker/tasklist model is that the number of processors used is bounded by the number of workers rather than by the number of tasks.

Although the worker model eliminates the overhead due to competing runnable threads, it does not avoid the overhead of division of labor. Tasks are created and added to the list even when there are more tasks than idle processors.

4.2.3 Reducing the Number of Tasks

The previous strategy could be amended so that each worker thread checks the size of the task list before adding a task. If the size of the list is less than the number of idle workers, then the task is created and added. Otherwise, the worker does the task directly.[1] Thus, the overhead of creating a task is incurred only when the task will indeed be performed in parallel.

The disadvantage of this strategy is that opportunities for concurrency may be lost. The decision to execute a divisible task in parallel is made before the first subtask is started. If, at that time, all workers are busy, then the task is performed serially. However, should a worker become idle before the first subtask is completed, it will not be able to perform the second subtask.

4.2.4 WorkCrews

The strategies of the previous two sections demonstrate that there is a conflict between using all processors and avoiding unproductive overhead. Both strategies minimize scheduler overhead by using one thread per processor. The former maximizes parallelism, but it fails to avoid the overhead of dividing a task when there are more tasks than workers. The latter minimizes overhead, but it sometimes leaves processors idle when tasks could be subdivided.

The reason the latter method misses opportunities for parallelism is that it decides whether to execute the second part of a task concurrently only on the basis of the number of busy coworkers when the task is begun. If a coworker becomes available before the worker completes the first part of its task, then concurrency is needlessly lost. The technique that I use in PTCC avoids this situation by allowing the second subtask to be started at any time during the execution of the first subtask. Each worker merely notes opportunities for concurrency without fully dividing tasks. If

[1] Forking only when there are idle processors is equivalent to this strategy.

the second part of a task has not been started by a coworker when the worker completes the first part, then it executes the second part serially.

In PTCC, WorkCrews are used to implement an extension and variation of the original worker/tasklist strategy with the following properties:

- Coarser grains of parallelism are favored over finer grains of parallelism.
- In the absence of idle workers, serial execution is favored over parallel execution. This is accomplished by
  - including as much of the overhead of division of labor as possible in the second subtask of a task
  - allowing workers to cancel unstarted tasks that they entered into the task list.
- Precedence constraints may be imposed between tasks.

Coarser grains of parallelism are favored because they tend to have a lower percentage of overhead than finer grains of parallelism — e.g., a parallel compiler should avoid statement-level compilation until procedure-level compilation fails to utilize all processors. To distinguish different grains of parallelism, a tree-structured task hierarchy is used. A task is divisible if it can be broken into subtasks that can be performed in parallel.

Workers always begin by executing their tasks serially — i.e., they traverse the task tree in depth-first order. When a worker encounters a fork in the tree, it adds a task to compute the right branch and proceeds to compute the left branch. The worker does the *minimum* amount of work required to define the task. Whenever possible, the overhead required to divide the task is included in the task itself. When it is finished evaluating the left branch, the worker attempts to cancel the right branch task. If the task has not been started by a coworker, this succeeds and the worker proceeds to evaluate the right branch serially without performing the additional overhead required for parallel execution. When a worker completes its task, it begins the coarsest granularity task of a coworker.

Figure 4.1 is an example of a task tree for a compilation. At the outset, shown in (a), the task is to compile a list of procedures (PL). It is being performed by worker w_1.

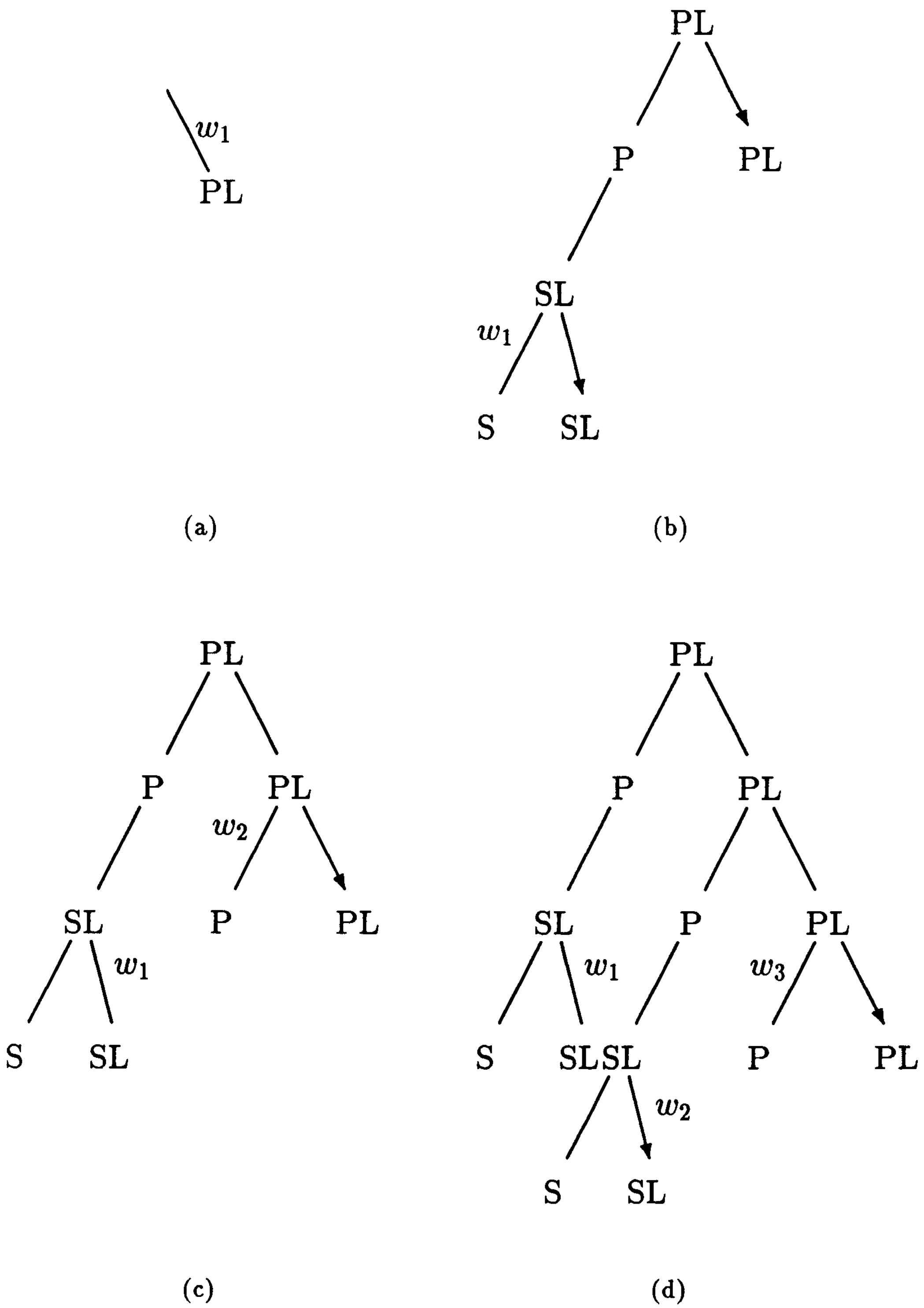

Figure 4.1: An example task tree.

In (b), w_1 divides the task into two parts: compiling a procedure (P), and compiling the remainder of the procedure list. An arrow is used to denote an unstarted task (PL). It proceeds to do (P) by reducing it into the task (SL) – compiling a list of statements – and divides the (SL) into the two parts (S) and (SL).

At this point, imagine worker w_2 decides to help w_1, so it begins (PL). It cannot begin (SL) because (PL) is the coarsest uncompleted subtask of w_1. Worker w_2 then divides its task. Meanwhile, assume w_1 finishes compiling its statement, so it cancels the previous (SL) task it created and performs the task itself. Both events are shown in (c).

Finally, in (d), w_2 has divided its task, and w_3 is helping w_2. At this point, suppose w_1 completes. It may choose to help either w_2 by performing (SL) or w_3 by performing (PL).

For PTCC, I wrote a module called WorkCrew to implement this strategy. The task tree and task list are represented using a deque (double-ended queue) of tasks for each worker. Conceptually, the task list is the union of the deques. Successive entries in a worker's deque denote finer subdivisions of its original task. A worker always adds tasks to the end of its task deque. When it finishes its current task, it removes a task from the end of its deque; this LIFO ordering is equivalent to serial execution. If the deque is empty, it removes a task from the front of one of the deques of its coworkers. This is equivalent to parallel execution at the coarsest granularity.

Because the abstract task list is decomposed into multiple worker task deques, synchronization costs are reduced. If a single, global task list were used, there would be contention between workers for the task list. With WorkCrews, each worker will make the majority of accesses to its task deque. Contention can occur only when an idle worker examines a coworker's task deque in search of work.

Managing Idle Workers

When an idle worker examines all of the workers' task deques, it may find that they are all empty. In this case, it must block until a coworker creates a new task. These semantics can be implemented using the `Condition` facilities of the Modula-2+ `Thread` module. When an idle worker is unable to find work, it calls `Thread.Wait` on the condition variable helpWanted. When a worker adds a task onto its deque, it calls Thread.Signal to awaken coworkers suspended on the helpWanted condition.

WorkCrews improve the strategy above by reducing the number of calls to Signal. A WorkCrew maintains a counter, unemployed, of the number of idle workers. Workers signal helpWanted only when unemployed is nonzero. Write access to unemployed is controlled by a global mutex. However, read access is unprotected because serializing the reads would also serialize task creation. Because aligned memory reads and writes are atomic on the Firefly, there is no danger that a read will corrupt a write.

A read may yield an out-of-date value of unemployed, but this is unlikely. Therefore, most of the unproductive calls to Signal are avoided, and few of the productive ones are lost. The latter is not a serious problem because the next worker to create a task will signal helpWanted, and tasks are created frequently. In the worst case, the worker which created the subtask will perform it serially after completing the sibling subtask.

Supporting Precedence Constraints

One limitation of the task list methods described so far is that they lack facilities for establishing precedence contraints between tasks. In the thread-oriented schemes, precedence constraints are implemented using mutexes. In worker/tasklist schemes where there is one worker per processor, this simple strategy is not acceptable since a blocked worker represents an idle processor. If tasks cause workers to block for any length of time, processor utilization will suffer.

WorkCrews can be extended to support precedence constraints between tasks by maintaining a precedence constraint tree. Define a *trigger* to be any task that represents a precedence constraint: if task A must be completed before task B may begin, task A is a trigger for B. When a worker begins evaluating a trigger, it adds a node for the trigger to the precedence constraint tree. This node contains a description of B and a count (initially one) of the number of workers evaluating tasks which are descendants of A. Whenever a worker begins a subtask of a trigger (or descendant thereof), it increments the count; when it completes the task, it decrements the count. The worker that decrements the count to zero is responsible for adding B to the task list.

4.3 Using WorkCrews in PTCC

This section describes the user interface of WorkCrews and how WorkCrews were used in PTCC.

A WorkCrew is a scheduling abstraction for parallel programs. Users of WorkCrews specify:

- the maximum number of threads to use
- precedence constraints between tasks
- how to divide tasks

4.3.1 Operations on WorkCrews

Creating and Joining WorkCrews

WorkCrews[2] are created and joined using the operations:

```
PROCEDURE Create(n:  INTEGER): WorkCrew.T;
PROCEDURE Join(crew:  WorkCrew.T);
```

`Create` returns a WorkCrew with `n` worker threads. `Join` suspends the caller until all of `crew`'s tasks have been completed.

Adding Tasks

Tasks are described as a procedure/argument pair. The procedure must be of type `Doer`, defined as

```
Doer = PROCEDURE(WorkerInfo, REFANY)
```

[2] The convention in Modula-2+ is to use the name T for the prinicipal type exported by a module. Hence, a WorkCrew object is of type WorkCrew.T.

The REFANY is the argument provided by the client. The WorkerInfo is provided by WorkCrew. It contains private data of the worker performing the task. When a client wishes to divide a task, it must pass this WorkerInfo to the relevant WorkCrew operations.

New, top-level tasks are added using the AddTask operation with the following interface:

```
PROCEDURE AddTask(
    crew:   WorkCrew.T;
    part1:  Doer;
    data1:  REFANY;
    part2:  Doer := NIL;
    data2:  REFANY := NIL)
  RAISES JoinInProgress;
```

After AddTask returns, crew first completes

part1(*some WorkerInfo*, data1);

and then completes

part2(*some WorkerInfo*, data2);

The tuples <part1 data1> and <part2, data2> represent two tasks. The second task is not started until the first one is finished: part2 will not be called until all workers performing some subtasks of part1 have finished.

If part2 and data2 are not supplied, then only a single task is added.

Calls to AddTask are atomic. All <part1,data1> tasks are started (but not necessarily finished) in the order they are added. This is used to avoid deadlock when task B needs data generated by task A but there is no reason to delay starting B until A completes.

AddTask is also atomic with respect to the invocation (but not the return) of Join. If AddTask is called after Join, then the AddTask operation fails and it raises the exception JoinInProgress.

Figure 4.2 describes how these procedures might be used to initiate a parallel compilation. Procedure ProcedureList compiles a list of procedures. PrintCompilerSta-

```
PROCEDURE StartUp (input:  Rd.T; output, error:  Wr.T);
  (* Compiles the sequence of procedures in input. *)
  (* Object code is written to output.              *)
  (* Errors are written to error.                   *)

  VAR
    cs:  CompilerState.T;
    wc:  WorkCrew.T;
  BEGIN
    cs := CompilerState.Create(input, output, error);
    wc := WorkCrew.Create(System.NumberOfProcessors);
    WorkCrew.AddTask(wc, ProcedureList, cs,
                     PrintCompilerStatistics, cs);
    WorkCrew.Join(wc);
    CompilerState.CloseOutputWriters(cs)
  END StartUp;
```

Figure 4.2: Starting up a WorkCrew

tistics, which is invoked only after ProcedureList has completed, might output information such as the number of procedures compiled. Sometime after AddTask is invoked, a worker thread of wc will perform

```
ProcedureList(wi, cs);
```

The first argument wi is the WorkerInfo for the particular worker. The second argument cs is the CompilerState that was passed to AddTask.

After all workers cooperating to perform the ProcedureList task have finished, some worker will perform

```
PrintCompilerStatistics(wi, cs);
```

4.3.2 Operations on Worker Tasks

This section describes the operations used to manipulate the task of a worker. The operations differ from the previous ones described in that they take a WorkerInfo as an argument rather than a WorkCrew.T.

Dividing Tasks

Clients specify how a task may be divided using the procedures `RequestHelp` and `GotHelp`. Typically, these operations are used as follows:

```
WorkCrew.RequestHelp( description of second subtask );
do first subtask
IF WorkCrew.GotHelp(...) THEN RETURN END;
do second subtask
```

`RequestHelp` and `GotHelp` are analogous to BEGIN/END's; they delimit a region. If a coworker answers a request, it will do so while the worker is executing the code within this region, which represents one subtask of the worker's original task. The second subtask is performed by a helper or by the code executed when `GotHelp` returns FALSE, but not both.

WorkCrew allows `RequestHelp/GotHelp` pairs to be nested. Outer requests for help are always answered before inner ones. Thus, coarser grains of parallelism are favored over finer ones.

The headers of the procedures are as follows:

```
TYPE Divider = PROCEDURE(REFANY, VAR Thread.Mutex):  REFANY;

PROCEDURE RequestHelp(
    wi:   WorkerInfo;
    doer: Doer;
    data: REFANY;
    divideTask:  Divider);

PROCEDURE GotHelp(wi:  WorkerInfo):  BOOLEAN;
```

The latter three arguments of `RequestHelp` form a triple <doer, data, divideTask> that represents a task. If the request is answered, some worker thread will invoke

```
doer(wi, divideTask(data, mutex));
```

The first argument `wi` is the WorkerInfo of the worker providing the help. The argument `divideTask` is a procedure provided to compute the initial state of the helper from the argument data provided by the worker requesting help. The role of the argument mutex is explained later.

The crux of the WorkCrew abstraction lies in the conditional execution of divideTask. Because divideTask is invoked only if and when the task is divided, clients should do the minimal amount of work required to create data and have divideTask do the rest.

Workers that have requested help use the procedure GotHelp to determine if their request was answered. If it was answered, GotHelp returns TRUE. Otherwise, GotHelp cancels the request and returns FALSE. Naturally, the test for help received and the cancellation of the request must be atomic with respect to attempts to answer the request. If GotHelp returns TRUE, it does not mean that the helper has completed – only that it has begun.

Sometimes, divideTask needs a resource that the worker will destroy once GotHelp returns TRUE. Therefore, it is necessary to delay GotHelp until divideTask has finished using the resource. This is done using the argument mutex: GotHelp will block until divideTask releases mutex.[3] Furthermore, no later requests for help by the worker are answered until mutex is released.

The purpose of the divideTask procedure is to perform the overhead necessary to divide the task. Where there is division, there is often a need for union. Thus, WorkCrew provides the operation:

```
TYPE Merger = PROCEDURE(REFANY);

PROCEDURE Merge(
    wi:    WorkerInfo;
    proc:  Merger;
    data:  REFANY)
```

Merge should be invoked only by divideTask procedures. It causes the helper to perform

```
proc(data);
```

[3] The mutex passed to divideTask is part of the WorkerInfo of the worker which requested help. It is the one used to implement the atomicity of GotHelp. When divideTask is called, mutex has already been acquired.

when the Doer listed in the call to RequestHelp returns.

Figure 4.3 is an example of how to divide the task of compiling a list of procedures using WorkCrews. (The code to process statement lists is very similar.) The worker assigned to compile the list of procedures requests that a coworker compile all procedures except the first; then it compiles the first procedure by calling Procedure. If its request was answered, it is finished. Otherwise, it repeats the process for the remaining procedures.

Note how the overhead of task division is divided into two parts: the creation and initialization of ncs, and the code of ProcedureListPrep. If the WorkCrew has only one worker thread, then the overhead introduced to compile a list of procedures is only the WorkCrew startup time, the creation of ncs and, for each procedure, a call to RequestHelp, GotHelp, and CompilerState.CopyEssentials.

ProcedureListPrep is used to divide the task of compiling a list of procedures. It creates new output writers for the helper by splitting those of the helpee, initializes a new CompilerState, and advances the input pointer to the next procedure to be compiled. For correctness, each new writer created must eventually be closed. Therefore, ProcedureListPrep uses Merge to specify that CompilerState.CloseOutputWriters should be called once the helper returns. Thus, although ProcedureList does not close its output writers, they are closed implicitly when it returns if and only if ProcedureList was performed by a helper.

Recall that writers must be split *before* they are closed (see section 3.4.3). Since the writers of cs may be closed implicitly when ProcedureList returns, its return must be delayed until ProcedureListPrep has completed splitting them. This is done by calling CompilerState.SplitOutputWriters before releasing mutex.

Establishing Precedence Constraints

As already mentioned, AddTask provides a mechanism for introducing precedence constraints between top-level tasks. WorkCrews also allow the user to specify precedence constraints between divisible subtasks. The relevant procedures are:

```
    PROCEDURE EnterSubTask(wi: WorkerInfo);

    PROCEDURE SubTaskIsFinished(wi: WorkerInfo): BOOLEAN;
```

```
PROCEDURE WhenSubTaskDoneDo(
    wi:  WorkerInfo;
    proc:  PROCEDURE;
    data:  REFANY);
```

EnterSubTask and SubTaskIsFinished are associated procedures much like RequestHelp and GotHelp. EnterSubTask is called to delimit the start of the first task. SubTaskIsFinished is called to delimit its end. Any number of RequestHelp/GotHelp or EnterSubTask/SubTaskIsFinished pairs may be nested recursively within the first task.

If SubTaskIsFinished returns TRUE, then there are no other worker threads still computing part of the first task, either because they all completed or the task was never divided. In either case, the worker simply begins the second task.

If SubTaskIsFinished returns FALSE, then there may be other worker threads still processing parts of the first task. Therefore, the worker may not begin the second task. Instead, it adds a description of the second task using WhenSubTaskDoneDo.

Figure 4.4 shows how I used WorkCrew precedence constraints to implement switch statements in PTCC. The strategy is the same as that of figure 3.2: the information from the case arms is stored in a synthesized attribute stored in the CompilerState. If SubTaskIsFinished returns TRUE, then SwitchStatement generates the dispatch table immediately. Otherwise, it is necessary to delay generating the dispatch table until the switch body has been completely parsed. Note that the latter case incurs the overhead of creating a new CompilerState and splitting the output writers.

```
PROCEDURE ProcedureList (
    wi:  WorkCrew.WorkerInfo;
    rcs:  REFANY);
  (* Effect: Compiles a list of procedures. Does not close the output writers of rcs.  *)
  VAR
    cs, ncs:  CompilerState.T;
    wc:  WorkCrew.T;
  BEGIN
    cs := NARROW(rcs, CompilerState.T); (rcs is a CompilerState.T)
    ncs := CompilerState.New();
    WHILE NOT CompilerState.NoMoreInput(cs) DO
      CompilerState.CopyEssentials(cs, ncs);
      WorkCrew.RequestHelp(
         wi, ProcedureList, ncs, ProcedureListPrep);
      Procedure(wi, cs);
      IF WorkCrew.GotHelp(wi) THEN RETURN END;
      END;
    END ProcedureList;

  PROCEDURE ProcedureListPrep (
      rcs:  REFANY;
      VAR m:  Thread.Mutex)
     :  REFANY;
    VAR
      cs:  CompilerState.T;
    BEGIN
      cs := NARROW(rcs, CompilerState.T);
      CompilerState.SplitOutputWriters(cs);
      Thread.ReleaseMutex(m);
      CompilerState.FullyInitialize(cs);
      SkipProcedure(cs);
      WorkCrew.Merge(wi, CompilerState.CloseOutputWriters, cs);
      RETURN(cs);
    END ProcedureListPrep;

  PROCEDURE Procedure(wi:  WorkCrew.WorkerInfo; cs:  CompilerState.T);
```

Figure 4.3: Compiling procedures concurrently using WorkCrews

```
PROCEDURE SwitchStatement(
    wi:  WorkCrew.WorkerInfo;
    cs:  CompilerState.T)
  VAR
    expr:  Expression;
    ncs:  CompilerState.T;
  BEGIN
    ASSERT(CompilerState.GetToken(cs) = TOKEN_SWITCH);
    ASSERT(CompilerState.GetToken(cs) = TOKEN_LEFT_PAREN);
    expr := Expression(cs);
    ASSERT(CompilerState.GetToken(cs.ts) = TOKEN_RIGHT_PAREN);
    CompilerState.EnterSwitch(cs);

    (* Call Statement to compile the body of the switch, which is      *)
    (* usually a compound statement.                                   *)
    (* When the body has been compiled, generate the dispatch table.   *)

    WorkCrew.EnterSubTask(wi);
    Statement(wi, cs);
    IF WorkCrew.SubTaskIsFinished(wi)
       THEN
         GenerateDispatchTable(cs, expr);
         CompilerState.ExitSwitch(cs);
       ELSE
         ncs := CompilerState.Copy(cs);
         CompilerState.SplitOutputWriters(cs);
         CompilerState.StoreSwitchExpr(ncs, expr);
         WorkCrew.WhenSubTaskDoneDo(wi, GenDispT, ncs);
    END;
    CompilerState.ExitSwitch(cs);
  END SwitchStatement;

PROCEDURE GenDispT(cs:  CompilerState.T);
  BEGIN
    GenerateDispatchTable(cs, CompilerState.GetSwitchExpr(cs));
    CompilerState.CloseOutputWriters(cs);
  END GenDispT;
```

Figure 4.4: PTCC's handling of C's switch statement.

Chapter 5

Performance of PTCC

To measure the performance of PTCC, I compiled an arbitrary set of source files on an otherwise idle five-processor Firefly. I varied both the granularity of parallelism and the number of worker threads used.

5.1 Versions of Compiler

TCC and PTCC are actually sets of compilers. Each successive version of PTCC exploits additional, finer grains of parallelism.

TCC

tcc1 The original TCC translated into Modula-2+.

tcc2 Same as tcc1, except all `POINTER` types converted to `REF` types.

PTCC

scanning Pipelines scanning with parsing and code generation.

procs Processes procedures concurrently.

bigstmts Processes `if`, `while`, `for`, do, `switch`, and compound statements concurrently.

anystmt Processes any statement concurrently.

Tcc2 is only interesting because of peculiarities of the Modula-2+ runtime system. In the Modula-2+ system, `REF` types, which are garbage-collected, are favored over `POINTER` types, which are allocated on a user-managed heap. Many library routines support only `REF` types. I found it necessary to replace all `POINTER` types with `REF` types in PTCC.

The Modula-2+ compiler handles `POINTER` and `REF` types differently, making `REF`s less efficient. Each time a `REF` is accessed, a runtime test is performed to check that it is non-NIL. Furthermore, a 4-byte header is created for each `REF` object. (Garbage collection, however, is not an issue because I did not enable the collector.) By measuring **tcc2**, these costs, which are unrelated to parallel compilation, can be factored out in the data analysis.

Scanning, **procs**, and **bigstmts** are actually variations of **anystmt** in which calls to the WorkCrew operations have been selectively removed (i.e., commented out). Each successive version adds finer grains of concurrency.

Scanning is the only version that incurs unnecessary overhead – namely, the overhead of:

- SplitWriters used to buffer the output of multiple compiler instances
- the indirection introduced by CompilerState objects
- the tree-structured symbol table
- matching of braces and parentheses.

5.2 Source Files

I ran each version of TCC and PTCC on several preprocessed files. The C preprocessor expands macros, strips comments, and inserts the text of each **#include**'d

	File	Number of lines		Procs	Brief description
		Original	PP		
Large	dispatch.c	1658	3053	3	Part of X Window System server
	eval.c	1127	1725	76	Pooh evaluator[a]
	mlcompile.c	1203	2936	12	Part of emacs
	origdisplay.c	1305	3827	18	Part of emacs
	parse.c	356	1833	15	YACC parser for Pooh
	schan.c	1372	4072	48	Part of emacs
	types.c	835	1254	64	Part of Pooh
	window.c	1251	2697	26	Part of X Window System server
	xterm.c	1153	3559	12	Part of X terminal emulator
Small	keywords.c	182	714	5	Part of Pooh
	libfns.c	—	451	3	Part of Pooh
	pooh.c	—	621	2	Part of Pooh

The *Original* column is the size (in lines) of the original source file.
The *PP* column is the size (in lines) of the preprocessor output for the file.
The *Procs* column indicates the number of procedures in the source file.
"—" indicates a missing data point.

[a] Pooh is a programming language developed by Eric Roberts for novice programmers. The files used here are from a Pooh interpreter.

Table 5.1: Source files used

file. The output of the preprocessor is typically longer than its input.

Table 5.1 lists a few parameters for each file. The files are divided into two categories: *small* and *large*. A small preprocessed file is approximately 100 lines long; a large one is approximately 1000 lines long. Parallel compilation of small files yields only minor reductions in elapsed time since small files require only a few seconds to compile.

5.3 Data

Appendix B contains the data obtained from running each compiler using from one to six worker threads for each input file. Data for four of the files also appears in tables 5.2 through 5.5. All runs were on an otherwise idle five-processor Firefly. Therefore, no performance improvement is expected when more than five workers are used.

The data in appendix B includes both the elapsed time and the CPU time for each compilation. To gauge the accuracy of these measurements, I ran **anystmt** ten times on the file *types.c* using one worker and ten times using four workers. The elapsed time measurements were very accurate[1] – only one sample differed for the one-worker case. For the remainder of this chapter, measurements of elapsed times are assumed to be accurate to within one second. The CPU time measurements were also low in variance. For the one-worker samples, the standard deviation was .29 and the mean was 29.2. For the four-worker samples, the standard deviation was .20 and the mean was 33.6.

5.4 Observations

For the large files, **anystmt** is 2.3 to 3.1 times faster than **tcc1** when five worker threads are used. If **tcc2** is used as the base of comparison, the range is 2.5 to 3.3. If one-second errors are assumed in the elapsed time, these ratios vary by roughly +/- .2. On the smaller files, the speedup is between 1 and 2.

Pipelined scanning reduces compilation time by approximately 10% using two workers. **Scanning** is .93 to 1.11 times faster than **tcc1** and 1.07 to 1.18 times faster than **tcc2** on the large files. Assuming one-second errors in the measurements of elapsed time, these ranges become .89 to 1.20 and 1.02 to 1.25 respectively. As expected, using more than two workers does not significantly reduce the time required for compilation. **Scanning** is only slower than **tcc1** on the file *parse.c*, which contains a large statically declared array. The slowdown is caused primarily by the substitution of REFs for POINTERs, which are used extensively by the code which handles initializers.

For many of the files (*eval.c*, *origdisplay.c*, *schan.c*, *types.c*, *window.c*, and *xterm.c*), processing procedures in parallel provides the bulk of the concurrency. For example, **tcc1** compiles *window.c* in 41 seconds (see table 5.2). With five workers **procs** takes 16 seconds. **Anystmt** only brings this down to 13 seconds. The data for *eval.c* is similar (see table 5.3). **Tcc1** requires 36 seconds, **procs** reduces this to 13 seconds, and **anystmt** provides no improvements.

Statement-level compilation does improve performance significantly for some of the

[1] The elapsed time measurements for all of the files almost never varied by more than one second.

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tccl	:41					
	tcc2	:43					
	scanning	:45	:39	:38	:39	:38	:39
	procs	:45	:24	:18	:16	:16	:16
	bigstmts	:45	:24	:18	:15	:14	:14
	anystmt	:44	:24	:18	:15	:13	:14
CPU Time	tccl	45.4					
	tcc2	47.3					
	scanning	50.0	51.1	50.4	51.1	50.5	50.7
	procs	50.6	51.5	53.0	54.2	55.2	55.3
	bigstmts	50.6	51.7	53.0	54.3	56.4	57.3
	anystmt	49.6	52.0	53.4	54.9	56.8	58.2

Table 5.2: Data for *window.c*

files (*dispatch.c*, *mlcompile*, and *parse.c*). For example, with five workers, **anystmt** requires only 22 seconds to compile *dispatch.c* (see table 5.4). **Bigstmts** requires 40 seconds. Another example is *mlcompile.c* (table 5.5). **Tccl** requires 33 seconds, **procs** reduces this to 23 seconds, and **bigstmts** requires only 13 seconds. These results are not surprising since the code portions of *dispatch.c* and *parse.c* are essentially one large **switch** statement. *Mlcompile.c* also has a couple of monolithic procedures.

Note that supporting statement-level compilation is never a performance liability when compared with procedure-level compilation. **Anystmt** is never significantly slower (more than one second) than **procs**. This suggests that the WorkCrew abstraction is making efficient use of parallelism.

Finally, note that the elapsed time is sometimes less than the CPU time in the one-worker column. For example, **tccl** compiles *dispatch.c* in 51 seconds, but it requires 56 CPU seconds to do so (see table 5.4). This is possible because the number of workers used is not a precise bound on the number of threads used. Topaz, the operating system for the Firefly, exploits concurrency. In particular, I/O is performed in parallel. Therefore, even the "serial" compilers **tccl** and **tcc2** use some concurrency.

		Number of worker threads					
	Compiler	1	2	3	4	5	6
	tcc1	:36					
	tcc2	:39					
Elapsed Time	scanning	:41	:35	:34	:34	:34	:34
	procs	:41	:23	:16	:14	:13	:13
	bigstmts	:42	:23	:16	:14	:13	:13
	anystmt	:41	:22	:17	:14	:13	:13
	tcc1	40.2					
	tcc2	43.2					
CPU Time	scanning	45.8	47.0	46.6	46.6	46.8	47.0
	procs	45.5	47.7	48.9	51.2	51.8	52.1
	bigstmts	46.8	48.1	49.8	51.5	53.3	53.6
	anystmt	45.8	47.8	50.8	52.4	54.2	56.0

Table 5.3: Data for *eval.c*

		Number of worker threads					
	Compiler	1	2	3	4	5	6
	tcc1	:51					
	tcc2	:56					
Elapsed Time	scanning	:57	:49	:49	:48	:49	:48
	procs	:56	:44	:44	:44	:44	:44
	bigstmts	:57	:41	:40	:40	:40	:39
	anystmt	:56	:31	:25	:22	:22	:21
	tcc1	56.4					
	tcc2	60.4					
CPU Time	scanning	63.0	64.2	63.7	63.6	63.7	63.4
	procs	62.2	63.5	64.2	63.5	64.2	64.0
	bigstmts	62.7	65.2	65.2	66.1	65.9	66.0
	anystmt	62.6	65.4	68.3	70.0	73.5	72.4

Table 5.4: Data for *dispatch.c*

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:33					
	tcc2	:35					
	scanning	:37	:32	:33	:33	:33	:33
	procs	:37	:23	:23	:23	:23	:23
	bigstmts	:38	:21	:17	:14	:13	:14
	anystmt	:37	:21	:16	:13	:13	:12
CPU Time	tcc1	36.8					
	tcc2	39.2					
	scanning	41.8	42.4	42.4	42.5	42.8	42.4
	procs	42.0	42.9	43.5	43.4	43.7	44.2
	bigstmts	42.6	42.6	44.1	45.4	46.6	47.0
	anystmt	41.5	43.1	44.4	45.5	47.0	47.6

Table 5.5: Data for *mlcompile.c*

5.5 Factors Limiting Performance

5.5.1 Lack of Parallelism

One performance limitation of PTCC is that it fails to keep all processors busy throughout a compilation. Table 5.6 lists the average processor utilization of **anystmt** for each of the C source files. These numbers were computed by dividing the CPU time by the elapsed time for the five-worker case. Note that compiling the small files tends to have lower utilization. This is probably because the start-up and clean-up times of the compiler process are independent of the source file.

Processor Utilization Graphs

When analyzing the processor utilization of a parallel program, it is helpful to have a utilization graph. A utilization graph displays the number of busy processors at different points during the execution of a parallel program. On the Firefly, this information is displayed by the watchtool, a program used to monitor system performance.

File	CPU Usage	Procs
dispatch.c	3.3	3
eval.c	3.9	76
keywords.c	3.2	5
libfns.c	2.9	3
mlcompile.c	3.6	12
origdisplay.c	3.7	18
parse.c	2.3	15
pooh.c	2.5	2
schan.c	3.5	48
types.c	4.4	64
window.c	4.4	26
xterm.c	3.4	12

Compiler is **anystmt** using 5 worker threads.
CPU Usage is CPU time divided by elapsed time.
The *Procs* column indicates the number of procedures in the source file.

Table 5.6: Processor utilization by file

Figure 5.1 is a utilization graph of the **anystmt** compiler processing each C benchmark file. Time is plotted on the horizontal axis, and the number of processors in use is plotted on the vertical axis.

As the graph indicates, PTCC is not always able to keep a five-processor Firefly fully utilized. In particular, the compiler performs somewhat poorly for the reasonably large files *dispatch.c*, *parse.c*, and *xterm.c*. The root of the problem is that PTCC processes declarations serially.

Dispatch.c begins with a lengthy sequence of `struct` and `typedef` declarations. During the early parts of the compilation of *dispatch.c*, two processors are used: one for the scanner, and one for a parser. Code is then reached before scanning completes, and processor utilization increases to the maximum of five. *Xterm.c* also begins with numerous declarations, and the compiler behaves in a similar manner.

Parse.c, a table-driven parser generated by YACC, contains two sections of code separated by large array declarations. The two peaks in the graph for *parse.c* correspond to the code sections. After the first code section has been processed, utilization drops

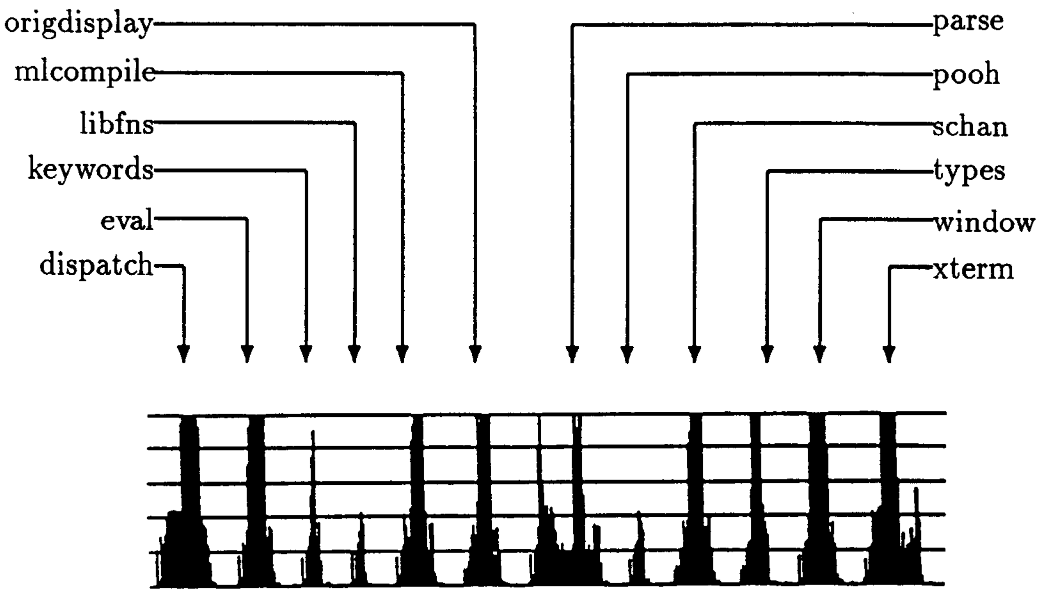

Figure 5.1: Processor utilization graphs for **anystmt**

to two. As for *dispatch.c*, one processor is scanning and one is parsing. However, here the scanner terminates before the parser finishes processing the arrays. Thus, utilization drops to one until the second section of code is reached.

Factoring Out Declarations and Scanning

With the measurements thus far, it is not possible to calculate the performance improvement that is due solely to the parallel processing of source code units. This improvement can be diluted by lengthy series of declarations, and it can be strengthened by the pipelined scanning. Therefore, I built a special version of **anystmt** that first scans its entire input. Then it measures the time it requires to process the file from the first procedure to the end of the file. For most source files, this has the effect of skipping the bulk of the declarations.

Table 5.7 contains the results of this experiment. Note that the performance for

ELAPSED TIME (SECONDS)
FROM FIRST PROCEDURE BODY TO END OF FILE

	Number of workers					Speed Up
File	1	2	3	4	5	[1] / [5]
dispatch.c	48	26	18	14	14	3.4
eval.c	32	17	13	10	9	3.6
keywords.c	5	3	2	2	2	2.5
mlcompile.c	29	15	12	10	8	3.6
origdisplay.c	28	14	10	8	7	4.0
parse.c	39	28	24	23	23	1.7
schan.c	26	14	10	8	8	3.3
types.c	19	10	7	6	5	3.8
window.c	36	19	13	11	9	4.0
xterm.c	32	17	12	9	9	3.6

Note: Time values are accurate to one second.
Speed Up column is simply the first column divided by the fifth.
The compiler used was a special version of **anystmt**.

Table 5.7: Peformance increase from concurrent processing of **code** units.

parse.c is still poor because *parse.c* has large array declarations after its first procedure.

5.5.2 Bus Contention

Another factor influencing the performance of PTCC is bus contention. The Firefly is a single, shared-bus multiprocessor. Only one processor may use the bus at a time, so processors may be delayed any time they need to access the bus.

Bus contention appears in the data as an increase in both elapsed and CPU times as the number of active processors increases. However, an increase in the number of active processors is typically accompanied by an increase in the overhead to divide work into smaller units. Therefore, only part of the increase in CPU time is caused by bus contention.

I used the Modula-2+ profiler to measure bus contention in PTCC. The profiler collects PC samples as a program executes. Using these samples, it determines how much CPU time was spent in each procedure of the program.

For a given source file, several procedures of PTCC are called the same number of times with the same arguments regardless of the number of worker threads used. These procedures perform a constant amount of work every time PTCC compiles the same program. However, the profiler indicates that these procedures require more CPU time as more processors are used. By measuring the discrepancy, it is possible to compute the slowdown in CPU speed caused by bus contention.

Table 5.8 contains the results of this experiment, which shows that bus contention slows PTCC by up to roughly 10%. The compiler was a profiled version of **anystmt**. The source file was constructed artificially to ensure that it contained sufficient opportunities for concurrency to keep four processors occupied. Because the profiler ties up roughly 40% of a CPU while it is gathering PC samples, a maximum of four worker threads are used. The experiment indicates that the processors run approximately 10% slower when roughly 4.4 processors are active than when a single processor is active.

Charles Thacker and Larry Stewart, the designers of the Firefly hardware, developed an analytical model of bus traffic on the Firefly for differing numbers of processors [15]. Their results project that the CPU slowdowns for two to five processors are .89, .87, .86, and .84. These slowdowns are significantly worse than those measured. They are based on the data referencing and instruction execution properties of typical programs running on the MicroVax II. I was unable to ascertain why the performance of PTCC was better than that of the typical programs studied by Thacker and Stewart.

5.5.3 Overhead to Support Concurrency

To support concurrency in PTCC, it was necessary to add mechanisms to control the sharing of data. The cost of these mechanisms is yet another factor which affects the performance of PTCC.

Number of workers	Mean Time	98% Confidence Interval		Slowdown [N]/[1]	98% Confidence Interval	
1	29.08	28.79	29.36			
2	29.67	29.43	29.91	.98	.98	1.00
3	30.15	29.82	30.48	.96	.94	.98
4	32.10	31.88	32.32	.91	.89	.92

Mean Time is the mean CPU time.
[N] is the mean CPU time for the *N*-worker case.
Slowdown is the is the speed of a processor when there are *N* active processors relative to the speed of a single active processor.

Table 5.8: CPU slowdown due to bus contention.

Synchronization Costs

Synchronization appears in several places in PTCC. The primary mechanism used to divide the compilation task was the WorkCrew abstraction. In turn, WorkCrew used the Modula-2+ Mutex and Thread facilities.

I used the Modula-2+ profiler to measure the cost of the WorkCrew and Thread procedures. The total cost of both abstractions when **anystmt** compiles *eval.c*, which allows extensive concurrency, is approximately .3 (+/− .04 at 98% confidence) seconds, which is insignificant.

Although WorkCrews and Threads probably account for the bulk of synchronization costs, some of these costs are scattered throughout PTCC. For example, procedures using Modula-2+'s `LOCK` statement have inline code to manipulate mutexes instead of calls to `Thread.Acquire` and `Thread.Release`.

Buffering (SplitWriter) Costs

When PTCC actually processes source code units in parallel, it must buffer the code generated by the parallel parsers. This is done using the SplitWriter abstraction. Individual measurement of SplitWriters indicate that they are 50% slower than ordinary writers.

Total Concurrency Costs

The total overhead introduced to support concurrency can be calculated by taking the difference between the CPU times of **tcc2** and **anystmt** in the one-worker case. The sum of these differences is 27.8 CPU seconds. This represents 6.7% of the 417.9 CPU seconds required by **tcc2** to compile all of the files.

The measurement above accounts for the overhead of using SplitWriters, WorkCrews and Threads, of duplicating CompilerStates, and of scanning concurrently[2] regardless of where this functionality was introduced in PTCC. Furthermore, the measurement does not include any significant bus contention overhead since the one-worker data was used.

The measurement does not account for increases in concurrency costs as additional processors are used because I was unable to distinguish between such costs and bus contention.

[2] Concurrent scanning, which can improve performance, is required to process source code units in parallel.

Chapter 6

Extensions

The algorithms presented in chapter 2 and implemented in PTCC are not powerful enough for production compilers. As described, they work only for one-pass languages, and they neglect syntax errors. This chapter sketches how the algorithms might be extended to allow syntax error recovery and to support multipass languages.

6.1 Handling Syntax Errors

In PTCC, syntax errors might be divided into two classes: those which involve delimiting terminals and those which do not. (See section 2.2.5 for a definition of delimiting terminals.) An error involves a terminal if the parser's recovery action for the error requires the insertion or deletion of the terminal.

If an error does not involve a delimiting terminal, no special error recovery measures are necessary for the parallel compiler: the error is contained within the region of a single parser which will detect and recover from the error independently. Otherwise, error recovery is more complicated.

Seshadri, Small, and Wortman are designing a parallel compiler whose scanner matches delimiters much like PTCC does [14]. They advocate that the scanner should correct any delimiter errors before parsing begins.[1] They argue that the

[1] Note that this precludes scanning and parsing in parallel.

scanner can recover from errors approximately as well as the parser can, especially when the source language uses distinct delimiters such as **if**, **endif**, **for**, and **endfor**.

This claim seems correct for insertion errors and substitution errors, where the patches are to delete or replace a token. However, for omission errors, which are patched by inserting a token, it is unclear that the scanner can find the proper place to insert the token. Furthermore, when an unpaired delimiter is encountered, the scanner has no context to determine whether the error is one of insertion or omission. The parser, however, has context to perform such recovery. For example, it can use the existence of a boolean expression following an **if** to determine whether an **endif** should be inserted.

I believe syntax error recovery is best performed during parsing. Error recovery can be added to PTCC using the following additional procedures:

- `SplitWriter.Truncate` is used to discard the output of any compiler instances working on parts of the source program following a syntax error. It takes a SplitWriter `sw` as an argument and discards all SplitWriters after `sw` in the SplitWriter list for `sw`'s target. `Truncate` requires that `sw` not have been closed; therefore none of the writes to the discarded SplitWriters have been flushed to the target when `Truncate` is called.

- `SymbolTable.Restore` is used to remove entries made by compiler instances working on parts of the source program following a syntax error. `Restore` takes a SymbolTable `st` and a positionstamp `p` as arguments and removes any entries in `st` whose positionstamps are later than `p`.

- `TokenStream.MakePristine` is used to undo changes made by compiler instances working on parts of the source program following a syntax error. `MakePristine` restores a TokenStream to its original form as created by the scanner (i.e., it undoes any changes made by parsers for syntax error recovery).

- `WorkCrew.AbortSuccessors` is used to halt and reclaim workers processing parts of the source program following a syntax error. `AbortSuccessors` is called by a worker to abort any tasks in progress which would have been performed after the worker's current task if only serial execution were used. Thus, it aborts any task which occurs later than the worker's task in a depth-first traversal of the task tree. Its signature is:

```
PROCEDURE AbortSuccessors(wi:  WorkCrew.WorkerInfo);
```

- WorkCrew.Restart is called by a worker which previously called AbortSuccessors. It causes the aborted tasks to be started anew. Its signature is:

```
PROCEDURE Restart(wi:  WorkCrew.WorkerInfo);
```

Given these procedures, error recovery can be implemented as shown in figure 6.1. Whenever a parser detects a syntax error involving a delimiter token, it first calls PrePatch, then it implements a patch, and finally it calls PostPatch. PrePatch stops potentially errant compiler instances and restores the SymbolTable and Split-Writers to a known state. Implementing the patch includes repairing the delimiter tree created by the scanner. PostPatch undoes any changes to the TokenStream made by errant compiler instances and restarts the aborted tasks.

6.2 Extending WorkCrews to Support Error Recovery

The error recovery mechanism described earlier relies on two new WorkCrew operations: AbortSuccessors and Restart. In designing these operations, it is imperative not to slow down the common WorkCrew operations such as RequestHelp and GotHelp. Because AbortSuccessors and Restart should not be called as frequently, their implementations are less critical.

6.2.1 AbortSuccessors

As mentioned earlier, WorkCrew.AbortSuccessors is called by a worker to abort any tasks in progress which would have been performed after the worker's current task if only serial execution were used. Thus, it aborts any task which occurs later than the worker's task in a depth-first traversal of the task tree. When AbortSuccessors returns, all successors have terminated execution.

Figure 6.2 is an example of a task tree. Each task is labeled with the worker which executes it. (The labeling represents only one of the possible divisions of the tree among workers.) For each worker, the Abort column of table 6.1 lists the coworkers that would be halted if it called AbortSuccessors.

```
PROCEDURE PrePatch (
    wi:  WorkCrew.WorkerInfo;
    cs:  CompilerState.T;
  );
  BEGIN
    WorkCrew.AbortSuccessors(wi);
    SymbolTable.Restore(CompilerState.GetSymTab(cs));
    SplitWriter.Truncate(CompilerState.GetOutput(cs));
    SplitWriter.Truncate(CompilerState.GetError(cs));
  END PrePatch;

PROCEDURE PostPatch (
    wi:  WorkCrew.WorkerInfo;
    cs:  CompilerState.T;
    ts:  TokenStream.T
  );
  BEGIN
    TokenStream.MakePristine(ts);
    WorkCrew.Restart(wi);
  END PostPatch;
```

Figure 6.1: Utility procedures for syntax error recovery in PTCC.

AbortSuccessors is, in some sense, an optimization. The successors may have already run to completion. Therefore, it serves to stop useless (and, in some applications, potentially infinite) computations.

WorkCrew does not attempt to undo aborted tasks. It is the responsibility of the client to do so.

Worker	Abort	Restart
w_1	w_2,w_3,w_4,w_5,w_6	w_2,w_3,w_6
w_2	w_5	w_5
w_3	w_2,w_4,w_5	w_2,w_4
w_4	w_2,w_5	w_2
w_5		
w_6	w_2,w_3,w_4,w_5	w_2,w_3

Table 6.1: Abort and Restart sets for figure 6.2

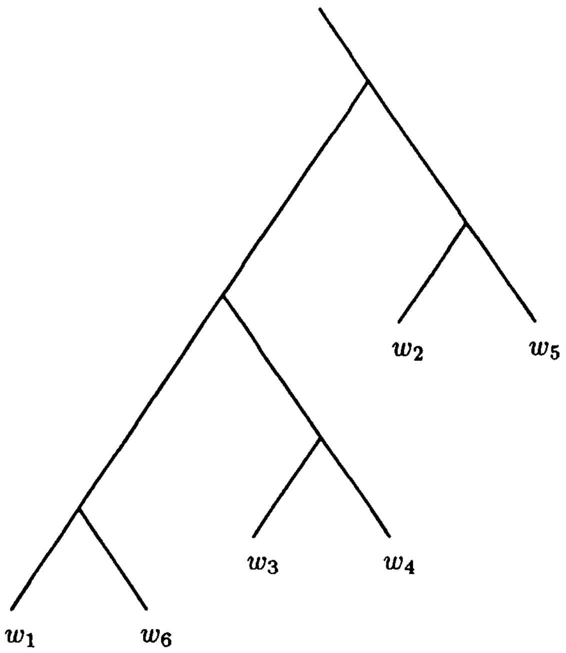

Figure 6.2: An example task tree.

Halting a Worker

Aborting a worker is a delicate operation because a running thread must not be halted at arbitrary times. For example, it may be holding a lock, or it may be in a critical section of code which temporarily violates data invariants.

One possible solution is to abort a worker only by raising an exception at the invocation of the next WorkCrew operation. Clients which do not invoke WorkCrew operations frequently might call a special `WorkCrew.CheckAbort` periodically. The cost of this strategy is an additional test for each WorkCrew operation, which seems acceptable. Upon receiving the `Abort` exception, clients would be required to restore any invariants and propagate the exception. This method of aborting workers is similar to the `Alert` mechanism of Threads described in [3].

Implementing `AbortSuccessors`

Define the relation **abort** between workers as follows: A **abort** B if and only if A precedes B in a depth-first traversal of the task tree. This relation is transitive and antisymmetric. It is total for any set of workers performing tasks in a common task tree.

To implement `AbortSuccessors`, the WorkCrew module must store enough information to be able to compute the **abort** relation for any pair of workers. One way to store the relation is to use a directed graph whose nodes are workers. An edge from A to B implies A **abort** B. Furthermore, since **abort** is transitive, a path from A to B also implies A **abort** B. Because the relation is antisymmetric, the graph is acyclic.

WorkCrew assigns each worker an id number from 1 to N, the number of workers. It maintains a set *Succ* of the ids of coworkers computing a successor of the worker's task. In terms of the graph, $j \in Succ_i$ if and only if there exists an edge from w_i to w_j.

An idle worker's *Succ* set is always empty. When a worker w_i helps worker w_j, it copies $Succ_j$ to $Succ_i$ since all successors of w_j's task are successors of w_i's new task. This is true because a worker's requests for help are answered in depth-first (FIFO) order. It then inserts i into $Succ_j$.

Given the *Succ* sets, it is possible to compute $Abort_i$, the set of workers aborted by w_i:

$$Abort_i = Succ_i \cup \bigcup_{k \in Succ_i} Abort_k$$

This computation is equivalent to determining all of the nodes (workers) reachable from node i in the directed acyclic graph representing the **abort** relation.

When w_i finishes its task, the information in $Succ_i$ must be transferred to the remainder of the graph. Thus, for each $Succ_k$ containing i, WorkCrew sets $Succ_k$ to be $Succ_k \cup Succ_i - \{i\}$. This removes the indirection through $Succ_i$ in the computation of *Abort*, so w_i may be reused.

Maintaining the *Succ* sets can be done efficiently using bit vectors. Only a constant amount of memory is needed.[2] Moreover, the sets are only updated when a task is performed in parallel, helping WorkCrews meet the goal of avoiding overhead when performing a task serially.

When w_i invokes **AbortSuccessors**, it computes $Abort_i$. Then, for each $k \in Abort_i$, it sets the **abort** flag in w_k's **WorkerInfo**. Thus, the next WorkCrew operation on w_k will raise the **Abort** exception. Worker w_i then waits for each aborted coworker to halt.

6.2.2 Restart

Once the situation requiring that a task's successors be aborted has been remedied, the successors must be selectively restarted. Not all successors are restarted since part of the remedy may include canceling a successor.

To provide this functionality, a new operation **Restart** is supported. **Restart** will create new instances of uncancelled aborted tasks which are immediate descendants of the aborting worker's task's ancestors. Table 6.1 lists the tasks that would be restarted if a worker in figure 6.2 called **AbortSuccessors** and **Restart** in succession.

A task is restarted in the same manner as that in which it was originally begun (see section 4.3.2). The WorkCrew module invokes its **dataPreparer** and **Doer**. There-

[2] The amount of memory needed is proportional to N^2, where N is the number of workers.

fore, clients must structure their programs accordingly. In particular, dataPreparer and Doer must not overwrite information needed to begin the task.

In terms of PTCC, this means doubling the calls to CompilerState.CopyEssential. Previously, CopyEssential was invoked once by the worker requesting help, and the helper used (i.e., overwrote) the resulting CompilerState. Now, dataPreparer must make and keep a private copy in case it is called multiple times.

After an abort, the number of tasks that might have to be restarted is bounded by the depth of the tree, which may be arbitrarily deep. Therefore, WorkCrew cannot store the information required to support Restart in constant space as it does for AbortSuccessors. Instead, it stores this information in the task tree. Previously, only the parts of the tree required to support precedence constraints between tasks were stored. This required one node per precedence constraint. Now, a node is also required for each task that is begun in parallel.

To restart the tasks that it aborted, a worker traverses the task tree from the root to its current task. It creates a new task for each aborted one. Because it starts at the root, the WorkCrew promise to answer requests for help in FIFO order is honored.

Between the invocations of AbortSuccessors and Restart, a worker can permanently cancel aborted tasks using GotHelp. GotHelp normally returns a boolean indicating whether a help request was answered. If it returns FALSE, no help was ever received so none was aborted. If it returns TRUE, help was received and thus aborted. When GotHelp returns, this help is then cancelled.

6.3 Compiling Multipass Languages

The algorithm of figure 2.3 can be extended for multipass languages in a way which preserves the number of passes required to compile the source language. The sequence of serial passes is replaced by a sequence of possibly parallel passes. Any pass whose input meets the requirements of sections 2.2.1 and 2.2.2 is made to run in parallel using the algorithms of figures 2.3 and 2.4. The passes might be allowed to run concurrently using pipelining; however, a pass may be forced to wait for information computed by earlier passes.

Typically, the front end of a compiler for a multipass language requires at most two passes: one which builds the parse tree and the symbol table from the input, and one which converts the parse tree into intermediate code. Since parse trees satisfy the constraints of section 2.2.2, the extended algorithm would produce two parallel passes for such compilers. Since parsing is faster than code generation, the latter pass should be delayed only when a variable is used before it is declared.

Note that the number of passes required is a function not only of the source language but also of the target language. For example, if the target language does not allow branches to symbolic labels, then the compiler must compute displacements based on the size of the code it generates. Thus, forward branches require two passes: one to compile the code between the branch and its target, followed by one to generate the forward branch instruction.

The desired code quality can also affect the number of passes. Suppose runtime efficiency concerns may mandate that registers (when available) be allocated for the index variables of loops.[3] If registers are scarce, then registers should be allocated for innermost loops first. This requires a separate pass for each loop. First, a register is allocated for the index variable of the innermost loop and code is generated for its body. Then, a different register is allocated for the enclosing loop's index variable and code is generated for its body, etc.

Such multiple passes can be implemented in the parallel compiler by maintaining a synthesized attribute *IndexRegs* for each **code** unit to represent the registers reserved for index variables nested within the **code** unit.

[3] Such optimizations are best left for a separate optimizer phase, but assume here that no such phase exists.

Chapter 7

Conclusions

The results of the experiment of chapter 5 indicate that parallel compilation can increase performance by factors as large as 3.3 on the five-processor Firefly. For the files listed in table 5.1, PTCC reduces the total compilation time from 350 to 152 seconds.

For parallel compilation to be useful, certain conditions must exist. First, there must be sufficient processors available. Given only one processor, PTCC is approximately 7% slower than TCC. With two processors, PTCC runs faster than TCC. Second, the source file must be sufficiently large to obtain significant reductions in elapsed time. PTCC runs 2.5 to 3.3 times faster than TCC on the larger files listed in table 5.1. For smaller files, the speedup factor ranges from one to two. This is not a serious problem since small files take less time to compile.

Procedure-level parallelism is usually sufficient to fully utilize a five-processor machine when PTCC is processing code (as opposed to declarations). For the set of C files discussed in chapter 5, statement-level parallelism was always sufficient. This suggests that PTCC could make use of additional processors, at least when compiling the files for which procedure-level compilation provided the bulk of the real concurrency. However, increased bus contention will dampen benefits from additional processors.

In PTCC, statement-level compilation is sometimes significantly better than, and never worse than, procedure-level compilation. This indicates that WorkCrews, whose overhead was insignificant, control parallelism efficiently.

The main factor limiting PTCC's performance is lack of parallelism while processing declarations. Mean processor utilization ranges from 2.3 to 4.4. A second factor is bus contention, which slows PTCC on the order of 10%. Finally, the overhead to support concurrency makes PTCC run approximately 7% slower using one worker than TCC.

Perhaps the best way to improve PTCC is to extend it to compile declarations in parallel. Processing statically initialized arrays concurrently is a very simple extension to PTCC. This would solve the performance problems when compiling files such as *parse.c*.

The bulk of declarations in large programs consist of type definitions, e.g., `struct` and **union** declarations in C. An area for future research might be to develop an algorithm which processes the bodies of `structs` and **unions** in parallel. Because the bodies of `structs` and **unions** contain information required to compile subsequent code, they are harder to process concurrently than **code** units.

Further research might also investigate the improvement obtained by integrating a parallel compiler with a syntax-directed editor. Such an integrated system has two benefits. The startup and cleanup times of the compiler are eliminated, so performance improves, especially on small files. Furthermore, since the input to the compiler is a syntactically valid parse tree, the compiler may be able to divide the tree evenly among processors before compilation begins.

Finally, little or no work has been done in the area of parallel optimization. However, advanced optimizers can take up to 50% or more of the total compilation time. Parallel optimizing compilers will have to divide the task of optimization to maximize increases in performance.

PTCC demonstrates that significant benefits can be obtained by exploiting concurrency during compilation. As tightly coupled multiprocessors become more common, so will parallel compilers.

Acknowledgments

I owe a great many thanks to the people at the DEC Systems Research Center for having me as an intern. Foremost, of course, is my host Eric Roberts, whom I want to thank for his technical assistance, his hospitality, and his patience with my writing. I'm also especially grateful to Chris Hanna for her abilities with ADB and for working diligently to get the Modula-2+ profiler working before I left. Many others helped make my stay in California a pleasant one: the volleyball team, fellow interns (Marks & Sparks), Tom, and Mark M. with his pizzicato bass-playing. I hope to work and play with you all again.

Upon my return to MIT, I was warmly welcomed into the ranks of the graduate community by my advisor John Guttag. I am indebted to him, as I am to Eric, for patiently reading and correcting my preliminary drafts on short notice. If this thesis is understandable, it is in a large part because of their efforts.

Appendix A

Thread.def

Modula-2+ provides several concurrency structures:

- Threads (type `Thread.T`) are lightweight processes in a shared address-space. Threads are manipulated using a fork/join semantics.
- Mutexes (type `Thread.Mutex`) are binary semaphores. Mutexes provide a mutual exclusion mechanism for threads.
- Conditions (type `Thread.Condition`) are used to signal events between threads. A condition may be waited upon, signalled (to some waiter chosen by the system), or broadcast (to all waiters).

The Modula-2+ `LOCK` statement is a useful shorthand used in conjunction with mutexes. In this report,

```
LOCK <expr> DO <stmt-list> END
```

can be considered equivalent to

```
Thread.Acquire(<expr>); <stmt-list>; Thread.Release(<expr>);
```

The actual semantics of the `LOCK` statement are more complicated due to exceptions. These complications are not important in this work.

The following is the definition module *Thread.def.* See [3] for a formal specification of threads, mutexes, and conditions, and a discussion of their use and implementation.

```
SAFE DEFINITION MODULE Thread;

  (**********************************)
  (*  Types and Related Constants  *)
  (**********************************)

  TYPE
    T = REF;

    ForkeeArg = REFANY;
    ForkeeReturn = REFANY;
    Forkee = PROCEDURE(ForkeeArg): ForkeeReturn;

  TYPE Mutex;

  TYPE Condition;


  (*******************************)
  (*  Thread Creation/Deletion  *)
  (*******************************)

  PROCEDURE Fork(proc: Forkee; arg: ForkeeArg): T;
    (* Fork a new thread at normal priority.
       The new thread will invoke proc(arg) *)

  PROCEDURE Join(t: T): ForkeeReturn;
    (* Wait until "t" has terminated, and get its result.  This can be
       done at most once for each "t".  If "Join" is never called, the
       result of a thread is discarded when the thread is
       garbage-collected. *)

  (********************************)
  (*  Synchronization Procedures  *)
  (********************************)

  (* Mutexes *)

  PROCEDURE InitMutex(VAR s: Mutex);
    (*  Initialize a Mutex (mutual exclusion lock).  Call once
    (*  Initialize a Mutex (mutual exclusion lock).  Call once
        before first use.  The initial state is "released".  *)

  PROCEDURE Acquire(VAR s: Mutex);
    (*  Acquire an exclusive lock.  If someone else has the lock,
        block until it is released.  *)

  PROCEDURE Release(VAR s: Mutex);
    (*  Release a previously acquired lock and if any threads
        are blocked awaiting the lock, wake up one of them.  *)


  (* Conditions *)

  PROCEDURE InitCondition(VAR c: Condition);
    (*  Initialize a condition variable.  Must be called
        precisely once, before first use of "c".  *)

  PROCEDURE Wait(VAR m: Mutex; VAR c: Condition);
    (* The caller releases the mutex "m" and waits on condition "c".  This
       combined operation is atomic with respect to calls of "Broadcast", so
       wake-ups cannot be lost.  Reacquires the mutex before returning.
     *)

  PROCEDURE Broadcast(VAR c: Condition);
    (* All threads that have executed "Wait" on the condition "c" cease
       waiting and become eligible to resume execution.  If no threads are
       waiting on "c", "Broadcast" has no effect.
     *)

  PROCEDURE Signal(VAR c: Condition);
    (* "Signal" is an optimization of "Broadcast".  It is functionally
       similar; however, it typically wakes up only one waiting thread.  This
       is not an absolute guarantee, since "Signal" will, on rare occasions,
       awaken more than one thread.  Thus, the client should use Signal only
       as a performance optimization in cases where it is unproductive to
       awaken multiple waiting threads.
     *)

END Thread.
```

Appendix B

Data

dispatch.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:51					
	tcc2	:56					
	scanning	:57	:49	:49	:48	:49	:48
	procs	:56	:44	:44	:44	:44	:44
	bigstmts	:57	:41	:40	:40	:40	:39
	anystmt	:56	:31	:25	:22	:22	:21
CPU Time	tcc1	56.4					
	tcc2	60.4					
	scanning	63.0	64.2	63.7	63.6	63.7	63.4
	procs	62.2	63.5	64.2	63.5	64.2	64.0
	bigstmts	62.7	65.2	65.2	66.1	65.9	66.0
	anystmt	62.6	65.4	68.3	70.0	73.5	72.4

eval.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:36					
	tcc2	:39					
	scanning	:41	:35	:34	:34	:34	:34
	procs	:41	:23	:16	:14	:13	:13
	bigstmts	:42	:23	:16	:14	:13	:13
	anystmt	:41	:22	:17	:14	:13	:13
CPU Time	tcc1	40.2					
	tcc2	43.2					
	scanning	45.8	47.0	46.6	46.6	46.8	47.0
	procs	45.5	47.7	48.9	51.2	51.8	52.1
	bigstmts	46.8	48.1	49.8	51.5	53.3	53.6
	anystmt	45.8	47.8	50.8	52.4	54.2	56.0

mlcompile.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:33					
	tcc2	:35					
	scanning	:37	:32	:33	:33	:33	:33
	procs	:37	:23	:23	:23	:23	:23
	bigstmts	:38	:21	:17	:14	:13	:14
	anystmt	:37	:21	:16	:13	:13	:12
CPU Time	tcc1	36.8					
	tcc2	39.2					
	scanning	41.8	42.4	42.4	42.5	42.8	42.4
	procs	42.0	42.9	43.5	43.4	43.7	44.2
	bigstmts	42.6	42.6	44.1	45.4	46.6	47.0
	anystmt	41.5	43.1	44.4	45.5	47.0	47.6

origdisplay.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:34					
	tcc2	:36					
	scanning	:38	:32	:32	:32	:32	:32
	procs	:38	:22	:17	:15	:14	:14
	bigstmts	:37	:21	:16	:14	:13	:13
	anystmt	:38	:21	:16	:14	:13	:13
CPU Time	tcc1	37.7					
	tcc2	39.8					
	scanning	43.1	42.7	42.7	42.8	43.1	42.8
	procs	42.4	43.1	44.0	45.0	45.9	45.6
	bigstmts	42.0	43.6	44.6	46.5	47.2	47.4
	anystmt	42.6	43.5	45.4	46.8	47.8	48.7

parse.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:42					
	tcc2	:48					
	scanning	:53	:45	:45	:45	:45	:45
	procs	:53	:44	:42	:41	:41	:41
	bigstmts	:54	:43	:40	:39	:39	:39
	anystmt	:53	:37	:32	:30	:29	:29
CPU Time	tcc1	44.7					
	tcc2	52.1					
	scanning	58.6	58.9	59.4	59.1	59.4	59.6
	procs	57.9	58.9	58.9	59.4	59.9	60.4
	bigstmts	58.7	58.8	60.0	60.9	61.0	61.2
	anystmt	58.4	61.3	63.8	66.7	67.6	68.2

schan.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:34					
	tcc2	:36					
	scanning	:37	:31	:31	:31	:31	:31
	procs	:38	:25	:21	:19	:18	:18
	bigstmts	:37	:21	:17	:14	:13	:14
	anystmt	:37	:21	:16	:14	:14	:14
CPU Time	tcc1	37.2					
	tcc2	39.6					
	scanning	41.6	42.2	43.0	43.1	42.8	42.9
	procs	41.7	43.0	43.7	44.7	45.7	45.8
	bigstmts	41.8	43.5	45.5	46.0	47.0	48.5
	anystmt	41.6	44.2	46.2	48.4	49.6	50.3

types.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:23					
	tcc2	:24					
	scanning	:25	:21	:21	:21	:21	:21
	procs	:25	:14	:11	:09	:09	:08
	bigstmts	:26	:14	:11	:09	:09	:09
	anystmt	:25	:14	:11	:09	:08	:09
CPU Time	tcc1	25.6					
	tcc2	27.2					
	scanning	28.9	29.6	29.2	29.6	29.3	29.4
	procs	28.9	30.6	30.7	31.5	32.3	32.8
	bigstmts	29.3	29.7	31.1	32.5	33.2	34.2
	anystmt	28.9	30.0	31.9	33.8	34.9	34.7

window.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:41					
	tcc2	:43					
	scanning	:45	:39	:38	:39	:38	:39
	procs	:45	:24	:18	:16	:16	:16
	bigstmts	:45	:24	:18	:15	:14	:14
	anystmt	:44	:24	:18	:15	:13	:14
CPU Time	tcc1	45.4					
	tcc2	47.3					
	scanning	50.0	51.1	50.4	51.1	50.5	50.7
	procs	50.6	51.5	53.0	54.2	55.2	55.3
	bigstmts	50.6	51.7	53.0	54.3	56.4	57.3
	anystmt	49.6	52.0	53.4	54.9	56.8	58.2

xterm.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:42					
	tcc2	:45					
	scanning	:47	:38	:38	:38	:38	:38
	procs	:48	:26	:22	:20	:19	:19
	bigstmts	:48	:26	:21	:18	:17	:18
	anystmt	:47	:26	:20	:18	:17	:17
CPU Time	tcc1	46.0					
	tcc2	49.2					
	scanning	52.0	52.8	52.9	52.5	53.1	52.5
	procs	52.5	53.6	54.2	55.0	55.6	56.1
	bigstmts	52.7	53.3	55.3	56.5	57.0	59.0
	anystmt	52.0	53.7	55.7	58.0	58.3	58.9

keywords.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:08					
	tcc2	:08					
	scanning	:09	:07	:07	:08	:08	:08
	procs	:09	:06	:07	:06	:06	:06
	bigstmts	:09	:06	:06	:06	:05	:05
	anystmt	:10	:06	:05	:05	:05	:05
CPU Time	tcc1	9.9					
	tcc2	10.5					
	scanning	11.6	11.8	11.8	12.2	12.0	11.8
	procs	11.5	11.7	12.7	12.1	12.3	12.3
	bigstmts	11.8	12.3	13.2	13.3	13.3	13.9
	anystmt	12.5	12.5	13.2	14.1	15.9	15.6

libfns.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:03					
	tcc2	:02					
	scanning	:03	:03	:02	:03	:02	:03
	procs	:03	:03	:02	:02	:02	:03
	bigstmts	:03	:02	:02	:02	:02	:02
	anystmt	:03	:03	:02	:02	:02	:03
CPU Time	tcc1	4.2					
	tcc2	4.3					
	scanning	4.7	5.6	5.0	5.2	5.2	5.6
	procs	4.8	5.7	5.1	5.4	5.6	6.0
	bigstmts	4.9	5.6	5.5	5.4	5.7	5.8
	anystmt	4.7	5.6	5.3	5.6	5.7	6.2

pooh.c

		Number of worker threads					
	Compiler	1	2	3	4	5	6
Elapsed Time	tcc1	:03					
	tcc2	:03					
	scanning	:04	:03	:03	:03	:03	:03
	procs	:03	:03	:03	:03	:03	:03
	bigstmts	:03	:03	:02	:03	:03	:03
	anystmt	:03	:03	:03	:03	:03	:03
CPU Time	tcc1	4.9					
	tcc2	5.1					
	scanning	5.5	6.2	6.3	6.2	6.1	6.1
	procs	5.3	6.1	6.1	6.0	6.3	6.5
	bigstmts	5.5	6.3	5.9	6.6	6.4	6.9
	anystmt	5.5	6.1	6.5	6.7	7.4	7.5

Bibliography

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Arvind, and R. Nikhil, and K. Pingali. I-structures: Data Structures for Parallel Computing. In *Lecture Notes in Computer Science*, Vol. 279, edited by G. Goos and J. Hartmanis, Springer-Verlag, 1987.

[3] A. D. Birrel, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. *ACM Operating Systems Review*, Vol. 21, No. 5, pages 94–102, November 1987.

[4] Hans-Juergen Boehm and Willy Zwaenepoel. Parallel attribute grammar evaluation. October 1986. Department of Computer Science, Rice University. Preliminary draft.

[5] Jacques Cohen and Stuart Kolodner. Estimating the speedup in parallel parsing. *IEEE Transactions of Software Engineering*, SE-11(1), January 1985.

[6] R. Cooper and K. Hamilton. *Preserving Abstraction in Concurrent Programming.* Computer Laboratory Technical Report 76, University of Cambridge, Cambridge, England, August 1985.

[7] James Frankel. *The Architecture of Closesly-Coupled Distributed Computers and Their Language Processors.* PhD thesis, Harvard University, 1983.

[8] Samuel Harbison and Guy Steele. *C: A Reference Manual.* Prentice-Hall, 1984.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[10] Daniel Lipkie. *A compiler design for multiple independent processor computers.* PhD thesis, University of Washington, Seattle, 1979.

[11] Dennis M. Mickunas and Richard M. Schell. Parallel compilation in a multiprocessor environment. In *Proceedings of the ACM Annual Conference*, pages 241–246, 1978.

[12] John A. Miller and Richard J. LeBlanc. Distributed compilation: A case study. In *IEEE Proceedings of the 3rd International Conference on Distributed Computing*, October 1982.

[13] Paul Rovner, Roy Levin, and John Wick. *On Extending Modula-2 for Building Large, Integrated Systems.* Research Report 3, Digital Equipment Corporation System Research Center, Palo Alto, California, 1985.

[14] Venkatadri Seshadri, Ian Small, and David Wortman. Concurrent compilation. To appear in *Proceedings of the IFIP Conference on Distributed Processing*, North Holland, 1988.

[15] Charles Thacker and Larry Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS II), October 1987.

[16] Mary-Claire van Leunen. *Modula-2+ User's Manual.* Digital Equipment Corporation Systems Research Center, Palo Alto, California, April 1986.

[17] David Wall. The Mahler intermediate language. June 1985. Digital Equipment Corporation Western Research Lab, Palo Alto, California. Unpublished.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock
Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301