SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(12), 1323–1348 (DECEMBER 1991)

An Investigation into Concurrent Semantic Analysis

V. SESHADRI* AND D. B. WORTMAN
Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada M5S 1A4

SUMMARY

Concurrency is an attractive method for reducing the execution time of compilers. By dividing source programs into segments which can be compiled concurrently, the task of compiling programs can be accelerated.

Many of the difficult problems which arise when constructing a concurrent compiler occur in the implementation of the semantic analyser. This paper investigates the problems involved in designing the semantic analyser for a concurrent compiler for a modern, block-structured language. Several approaches to solving the problems which arise are presented. These solutions are then implemented as part of a concurrent Modula-2+ compiler, running on a shared memory multiprocessor. A performance evaluation of these semantic analysers is presented.

KEY WORDS Compilation Concurrency Semantic analysis Modula-2 Concurrent compilation

INTRODUCTION

In many computing environments, a significant fraction of the time is devoted to compiling source programs into object code. Reducing the time spent in compilation would be directly beneficial to programmer productivity. As parallel processing technology becomes more commonplace, it becomes possible to contemplate designing compilers as parallel programs in order to achieve this goal.

Most compilers consist of a series of phases implemented in a varying number of passes, with each pass accepting as input the output of the previous pass.[1] The source program is the input to the first pass, and the final pass emits object code as its output. The most straightforward manner in which a compiler may be parallelized is to logically extend the multi-pass organization of the canonical compiler, by running each pass as a separate process in a pipeline with the processes communicating via shared data structures. This approach is limited in two ways. First, the compiler is not scalable with the number of processors available in the system, since the maximum number of processors which the compiler can use is equal to the number of passes in the compiler. Secondly, many compilers have an uneven run-time profile, spending a large fraction of their execution time in one or two passes. Thus, the

* Current address: Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, U.S.A.

0038–0644/91/121323–26$13.00

Received 8 June 1990
Revised 25 June 1991

speed of the pipeline will be constrained by the speed of the slowest pass of the compiler.

One may construct a parallel compiler which processes different portions of the source program through all passes of compilation concurrently. This would involve dividing the source program into disjoint segments early in the compilation process, processing these segments concurrently and producing object code for each individual program fragment, and finally combining the object code segments into a single executable program. This type of organization is more complex than a pipelined implementation since it raises implementation issues not encountered in a traditional compiler. First, the source code must be partitioned in such a manner as to introduce minimal overhead (i.e. communication and synchronization between compiler processes) while still preserving the ability to detect the same lexical, syntactic and semantic errors as a sequential compiler. Additionally, the code splitting should occur as early as possible in the compilation process so as to provide as much opportunity for parallel processing as possible. A third issue to address is the design of shared data structures (e.g. symbol tables), which must be constructed in such a way as to minimize contention when accessed by concurrent compiler processes.

When a concurrent compiler is constructed using the second approach described above, many of the difficult implementation problems arise in the design of the semantic analyser. This is because concurrent processing of different parts of the source program requires communication between the processes semantically analysing the different segments, whereas these same effects can be avoided in other phases of compilation by intelligent subdivision of the source program. This paper focuses on the design of the semantic analyser for this type of concurrent compiler.

CONCURRENT COMPILATION

The Concurrent Compiler Development Project at the University of Toronto [2-4] is an effort to use parallel processing aggressively to accelerate compilation. In this section we provide a brief overview of our project, including the types of source languages to which our compilation techniques apply, the multiprocessor architectures which we target towards, and the structure of the concurrent compiler which was implemented. In addition, a short survey of the previous work in the area is presented.

Source language and hardware

The source languages that we consider for compilation follow in the mainstream of modern programming languages. In particular, we require that they use Algol-60 or similar scope rules, and have reserved words rather than keywords which determine program structure. * Pascal, Ada and Modula-2 are examples of programming languages which meet these criteria.

Modula-2+ [5] is a modern, block-structured language which is derived from Modula-2. This language was chosen to be the source language for our concurrent compiler for several reasons:

* Reserved words may not be used by the programmer for other purposes, e.g. variable names, while keywords may be. The presence of keywords makes determination of program structure during lexical analysis much more difficult.

1. Modula-2+ is typical of the new generation of systems programming languages, containing features (e.g. modules) lacking in simpler languages such as Pascal, thus providing interesting challenges to the compiler writer.
2. The existence of a large library of Modula-2+ software provided a sizable test suite for our prototype compiler.
3. An existing Modula-2+ compiler, itself written in Modula-2+, could be parallelized instead of writing a concurrent compiler from scratch.

The concurrent compiler was written to operate on a shared memory MIMD multiprocessor with a small number of processors. The DEC Firefly multiprocessor workstation [6,7] served as our hardware platform. The Firefly used in this research is a symmetric multiprocessor consisting of five μVAX-II processors. These processors are connected via a shared bus to 16 megabytes of shared physical memory, with memory consistency being maintained by snoopy caches. The operating system [8,9] supports lightweight processes, making process creation a relatively inexpensive operation.

Compiler structure

Our approach to parallelizing a compiler is based upon the code-splitting approach described in the preceding section. A block diagram of our Modula-2+ compiler is given in Figure 1. The first column in the Figure describes the processing activity for definition modules (e.g. module stubs), the second column describes processing for the main module and the third column describes the processing of procedure scopes. It is important to note that the activities in these three columns share information from the compiler's symbol table. (See the section entitled 'Structure of the concurrent compilers' for a more detailed discussion.) Each pass of the compiler, except for the first and last, consists of a number of processes concurrently processing different segments of the source program. The number of concurrently executing processes is limited to the number of physical processors to prevent an unproductive explosion of processes. The compilation techniques described in this paper assume that there are many more compiler processes than physical processors on which to run them. An internal scheduler is required to determine the order in which code fragments arriving from the previous pass should be processed. Pipelining is also implemented between the passes.

The division of the source program is accomplished early in compilation by the lexical analyser and splitter. The lexical analyser transforms the source program into a sequence of lexical tokens. The splitter divides this token sequence recursively at the boundaries of major scopes (e.g. procedures, modules). In order to do so, the splitter must be endowed with some simple parsing functionality so that it may recognize scope boundaries. A simple parenthesis matching algorithm is sufficient, since the scopes of modern, high-level languages are delimited by distinct token pairs (e.g. begin/end). * The output of the splitter is a program tree with each node consisting of a sequence of tokens representing a scope in the program. Some tokens are placeholders, containing references to child scopes removed from their parents.

The division of the source program into its constituent scopes allows concurrent parsing to be performed on these scopes without requiring communication or synch-

* We assume that lexical analysis is *context-free,* i.e. independent of programmer declarations.

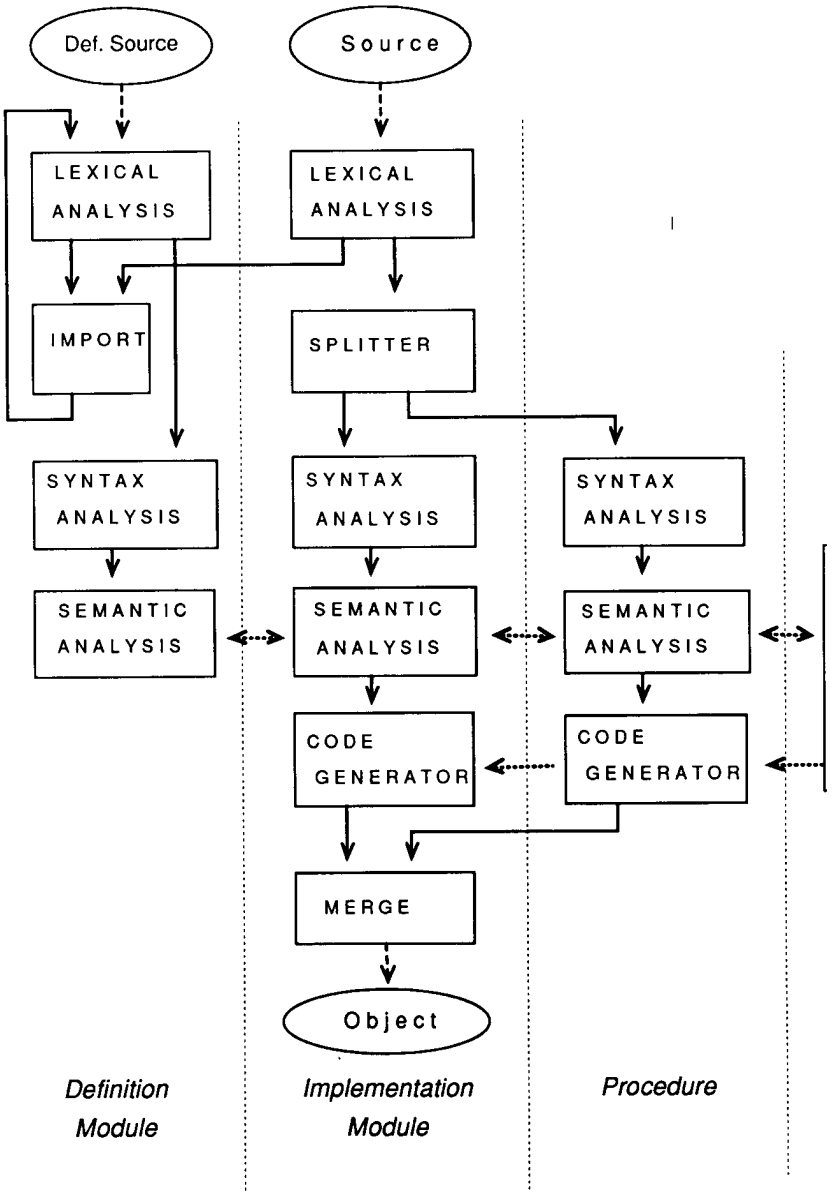

Figure 1. Concurrent compiler block diagram

ionization between the parsing processes. However, this is not true for semantic analysis. The Algol-60 scope rule [10] permits identifiers declared in one scope to be legally referenced in another, thus allowing information to flow between scopes. Hence, processes must communicate during this phase in order to properly conduct semantic checks on the declaration and use of identifiers.

Code generation can be performed on individual scopes without the need for interprocess communication, except for non-conflicting access to the compiler's

symbol table. Position-independent code can be generated for each scope with the individual object modules merged afterward.

Of paramount importance in compilation is error detection. The strategy of dividing a source program at scope boundaries allows easy detection of the lexical, syntactic and semantic errors in the program. Since the lexical analyser is sequential, it detects all errors that it would in a sequential compiler. In addition, it detects any syntactic errors related to nesting of scopes. If the nesting of scopes is syntactically correct, then all remaining syntax errors can be detected within the scopes themselves. Detecting semantic errors requires information transfer between processes due to the Algol-60 scope rule.

Previous work

Most of the theoretical work in parallel compilation has dealt with parsing regular and context-free grammars. In general, the proposed schemes involve a linear array of identical processing elements analysing arbitrary, disjoint segments of the source program. Fischer [11] was the fit to extensively study parallel parsing. Cohen *et al.* [12,13] extended Fischer's work by investigating the speed-up which was available by parsing concurrently. He showed that there was a large overhead in merging the output of individual processors (which is avoided in our compiler by the scope-level subdivision of the source program). A summary of parallel parsing research can be found in Reference 14; some recent work in this area appears in Reference 15. The only theoretical investigation into other phases of concurrent compilation has been the work of Schell, [16] who studied semantic analysis and code generation based on the parallel evaluation of attribute grammars.

The earliest practical attempts to parallelize compilation occurred in the early 1970s. [17-21] Most of these efforts involved the use of vector processing techniques in the compilation of Fortran. Later work in the 1970s involved pipelining compilation on loosely coupled multicomputers. [22-25]

Compiler designs based on division of the source program at well defined syntactic boundaries are most relevant to the work presented here. The earliest implementation reported was by Frankel. [26] Frankel's compiler was a modification of a one-pass Pascal compiler in which procedure bodies were compiled in parallel with the enclosing scope. Processes were created to handle these procedure bodies as they were encountered while processing the parent. His compiler ran on a network of Xerox Alto workstations, and achieved a speed-up of 3·7 on six processors. However, the compiler was built upon multiple levels of interpretation, making it more CPU bound than most compilers and thus optimistically skewing his results.

Vandevoorde [27] constructed a concurrent C compiler to run on a five processor DEC Firefly workstation. His compiler was partitioned into two passes—a lexical analyser and a syntax-directed translator. This compiler implemented a fine grain of parallelism—large statement lists were broken into smaller ones for concurrent processing. A speed-up of 2·3 to 3·1 on five processors was reported when compiling large programs, whereas a speed-up of 1·0 to 2·0 was reported for smaller programs.

Boehm and Zwaenepoe1 [28] experimented with parallel attribute grammar evaluation for semantic analysis. They implemented a combined static/dynamic attribute grammar evaluator for a subset of Pascal which ran on a network of SUN-2 workstations. All attributes local to a processing node were evaluated statically, while

attributes shared between nodes were evaluated dynamically. They achieved a speed-up of 2·5 on a five processor network when compiling a 1000-line program.

CONCURRENT SEMANTIC ANALYSIS

The purpose of semantic analysis is to verify that the source program obeys all the non-syntactic constraints imposed by the programming language. The semantic analyser in a compiler processes declarations and records information from these declarations in symbol and type tables which are used by subsequent phases.

The Algol-60 scope rule used by the languages under consideration here implies that semantic information about identifiers can flow between scopes—an identifier declared in outer scopes may be legally referenced in inner scopes. In our compiler model, with scope-level concurrency, the process semantically analysing the scope of declaration of an identifier will frequently not be the same as the one processing the scope of usage of that identifier. Thus communication between compiler processes is necessary during semantic analysis.

The information exchanged during semantic analysis is the identifier attribute data which is stored in the compiler's symbol tables, making these tables the media by which the processes communicate. Consequently, two major questions are raised when implementing a concurrent semantic analyser:

1. How does the need to support concurrency alter the basic structure of the symbol tables?
2. How should the compiler deal with missing or incomplete symbol table information?

These two issues are interrelated—in particular, solutions to the first significantly affect the second. In this section, we examine these issues and propose solutions to them.

Symbol table structure

The basic symbol table in a sequential compiler is a stack of (name, attribute) pairs. When a declaration is processed, the name and attribute of the declared identifier are pushed onto the top of the stack. Markers in this stack delimit scope boundaries. Looking up the attribute of an identifier in the symbol table consists of searching down from the top of the stack until the identifier is found. When the semantic analyser finishes processing a scope, entries for all identifiers which were declared in that scope, are popped off the stack. Identifiers are not visible outside their scope of declaration, and the most recent definition of an identifier is the one which is found by the symbol look-up routine. Variations of this technique are employed in most sequential compilers.[1]

The stack mechanism is unsuitable for a concurrent compiler. In sequential compilers, symbol table entries from scopes at the same nesting depth do not exist simultaneously on the stack. However, concurrent compilers require this feature in order to process scopes at the same nesting level in parallel.

Instead, a tree-structured approach is used in our compiler, with each scope in the source program allocated a separate hash table in which all declarations for that scope are recorded. Each hash table is protected by a mutual exclusion mechanism

to protect the table's contents against corruption by concurrent writes. The entire table is locked when a new entry is being added. Individual entries are locked when they are being modified. A look-up of an identifier referenced in a scope consists of searching outward from the symbol table of the scope of reference through the tables of parent scopes until the scope of declaration is found. Vandevoorde [27] used a similar scheme in his compiler.

The 'doesn't know yet' (DKY) problem

Most modern programming languages require that the programmer declare all identifiers before they are used in the program. Various mechanisms may be provided to allow restricted forms of use before declaration (e.g. constructing a pointer to an undeclared type). Compilers for languages that do not require declaration before use (e.g. PL/I) usually make a separate pass over the source program to collect declarations before semantic analysis. These considerations lead to an unstated invariant that exists in sequential compilers with respect to the symbol tables:

> *Table completeness invariant.* At any point during semantic analysis, the absence of a given identifier in the symbol table is a definitive indication that the identifier has not been declared.

This invariant does not necessarily hold in concurrent compilers, leading to a difficulty which we call the *'doesn't know yet'* (DKY) problem. The most common cause of the problem arises when an identifier is referenced in a nested scope, but is not declared in that same scope. A transient version of the problem arises when a symbol table entry is temporarily incomplete during its creation. For example, the symbol table entry for a large record type might require many symbol table operations to complete its definition. To locate the definition of a non-local identifier, a search through the symbol tables of the parent scopes is required. However, because the symbol tables are being built by separate processes, the failure to find a name in a parent symbol table does not necessarily mean that the name is not declared in that scope; its declaration may exist but not have been processed yet or its declaration may be incomplete. If either of these situations arise then the searching process doesn't know yet (DKY) whether or not a visible declaration of the identifier in that parent scope exists.

Our goal in designing concurrent compilers is to achieve faster compilation through concurrent processing of scopes. The DKY problem is the primary impediment to achieving this goal. Every time a compiler process waits for a DKY, overhead is incurred dealing with the DKY event and concurrent processing may be restricted if there are no other processes that can execute while the DKY event is being waited for. An example that demonstrates this problem is a module with a large number of declarations and a number of child procedures that all incur a DKY wait on the last declaration in the module.

The strategies described in the following sections take two approaches to minimizing the impact of the DKY problem. The avoidance strategy prevents DKYs from occurring. The handling strategies allow DKYs to occur but try to minimize their impact. The choice between these, and other, ways of dealing with the DKY problem is usually made based on tradeoffs between ease of implementation and level of processing speed-up achieved.

We consider three methods of dealing with the DKY problem. The first approach employs scheduling of scope processing during semantic analysis to eliminate the DKY problem altogether. The second method allows DKYs to occur, suspending processing when a DKY occurs. The third approach is a hybrid of the other two, splitting semantic analysis into two phases and allowing DKYs to occur during one but not the other.

DKY avoidance

The avoidance approach to solving the DKY problem schedules processes in the compiler in order to avoid DKYs during semantic analysis and thereby maintains the table completeness invariant. This technique was used by Franke1 [26] and Vandevoorde [27] in their compilers. In order to determine the scheduling constraints on semantically analysing a scope, it is necessary to examine the identifier attribute information flow and the types of scopes in the source language.

In the languages we consider, information flows between scopes in two ways:

1. A nested scope may inherit or explicitly import identifiers from its parent

2. A nested scope may export identifiers and their attributes (both explicitly and implicitly) to its parent scope.

This information flow determines the flow of semantic attributes between scopes and constrains the order in which semantic analysis must be performed if the required semantic information is to be available when needed. The imported and exported identifiers are the carriers of this semantic information.

Since scopes can inherit all the identifiers declared in their parent scopes, parent scopes must be processed before their child scopes. More precisely, semantic analysis of a scope cannot begin until all of the definitions in its parent scopes which it may legally inherit are entered into the symbol tables. This does not preclude concurrent processing of child scopes with their parents, since not all declarations in parent scopes are visible in child scopes (e.g. declaration before use rules restrict this). In order to determine precisely when parent scopes may be concurrently processed with their children, we must consider the manner in which declarations within a child scope are visible to the parent.

In the programming languages being considered, the scopes can be classified into the following categories:

(a) unnamed scopes such as begin/end constructs
(b) named scopes such as procedures, which have parameter lists containing type information
(c) named scopes such as modules, which export identifiers but do not specify the identifiers' type at the point of export.

Unnamed scopes cannot be referenced, and hence identifiers declared within them are not visible outside. This implies that concurrent processing of an unnamed scope with its parent can proceed after all declarations in the parent that are visible to the child have been processed. In our compiler, and in most sequential compilers, declarations in unnamed scopes are processed by putting these definitions in the symbol table of the enclosing procedure or module. The symbol table look-up

mechanism is modified to ensure correct application of the language's scope rules. This is usually more efficient than incurring the overhead associated with scope creation and deletion for each unnamed scope.

Procedure-like scopes are named (and hence can be referenced elsewhere) and have an associated parameter list. The identifiers in the parameter list are attributed with type information. The identifier attributes, and the name of the scope are usually the only information exported. The definition of parameter type attributes is usually done in the context of the parent scope. Therefore, the parameter list of the child scope must be processed before the remainder of that scope can be analysed in parallel with its parent. For example, in Modula-2+ the child and the parent can be processed concurrently once the parameter list has been analysed.

Module-like scopes export selected identifiers. In general, the exports list of module-like scopes contain simple identifiers, with attribute information being provided by declarations elsewhere in the module. References in outer scopes to these exported identifiers cannot be semantically analysed without the possibility of DKYs occurring until the declarations of these identifiers in the module-like scope have been processed and entered into the symbol table. Additionally, all declarations in the module's parent scope which can be referenced inside the module-like scope must be processed before the module can be processed.

We make the distinction between procedure-like and module-like scopes since the externally visible semantic attributes of procedure-like scopes are usually completely determined by declarations given in the procedure header. In module-like scopes the declarations for exported names may be embedded at *arbitrary points* within the scope. This embedding makes concurrent semantic analysis of module-like scopes much more difficult. However, this restriction does not preclude concurrent processing within the module-like scope itself. For example, procedure bodies within a module-like scope can be processed in parallel. Note that in some languages (e.g. PL/I) procedures are module-like scopes because the attributes of parameters are specified in declarations in the body of the procedure.

Given these restrictions on the concurrent processing of scopes, a DKY avoidance algorithm can be devised. This algorithm is given in Figure 2. It uses a 'parents before children' strategy in scheduling the semantic analysis of program scopes, starting with the outermost program scope. Note that the algorithm given in Figure 2 assumes declaration before use of identifiers in the program is required. The algorithm is easily extended to allow relaxations of this rule at some loss in potential concurrency.

By avoiding DKYs entirely, compiler table management is greatly simplified. Looking up the definition of an identifier consists of searching through the symbol tables of all containing scopes. If a definition of an identifier in a symbol table is not found, then no visible declaration of the identifier in that scope exists. Defining an identifier in a symbol table consists merely of adding its name and attribute to the proper hash table.

DKY handling

The major difficulty with the DKY avoidance approach is that it is conservative in permitting parallel processing. DKY avoidance presupposes that there are a large number of declaration dependencies between child scopes and parent scopes (i.e.

```
PROCEDURE SemAnal (scope)
    LOOP
        EXIT WHEN end of scope.
        Semantically process scope until either
            (1) a child scope is encountered or
            (2) scope is module-like and all its exported
            identifiers have just been processed.
        IF case (2) THEN
            Signal process handling scope's parent to continue.
        ELSIF a module-like child scope has been encountered THEN
            Fork a processes to do SemAnal on that scope.
            Wait until it has finished processing all its exported identifiers.
        ELSIF a procedure-like child scope. has been encountered THEN
            Semantically process its parameter list.
            Fork a process to do SemAnal on that scope.
        ELSE (*A nameless child scope has been encountered *)
            Fork a process to do SemAnal on that scope.
        END IF
    END LOOP
END SemAnaI
```

Figure 2. DKY avoidance algorithm

child scopes extensively reference declarations in parent scopes). An alternative approach to dealing with the DKY problem is to permit processing of a scope to continue until a DKY arises, suspending processing until the DKY can be resolved.

This approach is opportunistic, in that it does not use any conservative scheduling. However, it does complicate the compiler's symbol table management routines. This complication arises from

(a) the need to support incomplete symbol table entries
(b) the need to detect and resolve DKYs.

Incomplete entries

Some types of identifier definitions require multiple symbol table accesses and considerable processing to create. While these entries are being constructed they are incomplete. If other processes look up these identifiers while they are incomplete, the look-up must be suspended until the entry has been completed. Therefore, entries in the symbol table must be augmented with a field to denote the completeness of the entry and a lock with a queue on which processes can wait for the entry to be completed.

There are several ways in which incomplete entries can arise. Some language constructs such as record types, enumerated types and forward declared types require many symbol table accesses to complete and thus result in one type of incomplete entry. For these entries, the process creating the locked entry and finally completing and unlocking it are the same. Another type of incomplete entry arises from the way in which scopes are separated in the source program for concurrent processing. These scopes are replaced in their parent scope by placeholder tokens containing the identity of the child scope. When a semantic analysis process encounters such a token, it creates an incomplete, placeholder entry for it in its own symbol table. This entry is eventually overwritten with a concrete entry (and completed) by the process semantically analysing the child scope, which contains the complete attribute

information about its name. This type of incomplete entry is completed by a different process than the one which originally entered it into the symbol table.

Detecting and resolving DKYs

DKYs can occur in two ways:

1. A look-up of an identifier in a symbol table finds an incomplete entry.
2. A look-up of an identifier in a symbol table finds no entry at all, but the table is incomplete. This is by far the most common case.

DKYs on incomplete entries can be resolved in two ways—either the entry can be completed by the same process which created it, or it can be completed by another process. The first type of resolution usually corresponds to the completion of a complex definition (e.g. a type declared forward, a record type or an enumerated type) in the local scope. The second type of resolution occurs on incomplete entries which correspond to named child scopes that are entered into the symbol table of their parent scope. A concrete entry (with complete attribute information for the child) is entered into the symbol table of the parent scope overwriting the entry generated from the placeholder token. In the performance results section, we distinguish these two types of incomplete DKYs as *local* DKYs and *placeholder* DKYs.

DKYs on incomplete symbol tables can be resolved in two possible ways. First, the identifier may eventually be defined at which point any processes waiting for its definition can be unblocked. The second way in which this type of DKY can be resolved is by having the table completed without a declaration of the identifier occurring in that scope. The process semantically analysing a scope is responsible for unblocking the appropriate suspended processes when that scope's table is complete. This type of DKY should not occur when a process is searching through the symbol table of the scope which it is processing—this would cause that process to block forever.

Given these criteria, procedures for entering identifiers into symbol tables and looking up identifier definitions in the symbol tables can be developed. A sample skeleton procedure for performing identifier definitions in a symbol table is given in Figure 3, and the corresponding procedure for performing look-ups is given in Figure 4.

Two part semantic analysis

Two basic types of constructs exist in high-level programming languages—declarations and statements. Semantic analysis of declarations requires both reading from and writing to the symbol tables, whereas the semantic analysis of statements usually requires only reading from the tables.

In order to eliminate the effects of the DKY problem as soon as possible, it is beneficial to complete the compiler's symbol tables as quickly as possible. Since only declaration analysis adds information to the tables, the table for a scope can be said to be complete as soon as all the declarations in that scope have been processed. Since we assume that there are a limited number of processors available to execute the compiler processes, the completion of all symbol tables may be accelerated by deferring the semantic analysis of statements until all declarations have been pro-

```
PROCEDURE Define (identifier, scope)
    Lock semaphore on symbol table of scope.
    IF modifying definition of incomplete entry THEN
        Modify the entry.
        (* Includes overwrite of placeholder entry with concrete entry *)
    ELSIF making entry for placeholder token but concrete exists THEN
        (*Do nothing *)
        Release semaphore on symbol table of scope.
        RETURN
    ELSE (* Normal case *)
        Create entry for identifier in the symbol table.
    END IF
    IF entry is complete THEN
        IF entry is locked THEN
            Unlock the entry.
        END IF
        Release semaphore on symbol table of scope.
        Signal all processes blocked on a definition of identifier.
    ELSE (* For incomplete entries do the following *)
        Lock the entry.
        Release semaphore on symbol table of scope.
        RETURN
        (* Entry will be completed and unlocked by subsequent processing*)
    END IF
END Define
```

Figure 3. Identifier definition routine for DKY handling

cessed. Two part semantic analysis is most useful in dealing with nested scopes because it helps reduce the occurrence of DKYs from inner scopes on symbols declared in outer scopes. A procedure containing a few local declarations, several nested subprocedures and a large number of statements is one example where the performance of two part semantic analysis would be better than one part semantic analysis.

In two part semantic analysis, the parser is used to divide a scope into its declarations and statements. The first part of the semantic analysis phase is declaration analysis, during which the declarations of the program are processed and entered into the symbol tables. After the tables are built, the statements of the source program are processed.

For high-level languages which permit declarations and statements to be syntactically intermixed (e.g. Turing [29]) significant benefits may accrue by using this strategy. In a one part semantic analyser which employs DKY avoidance, child scopes have to wait for all statements in addition to the declarations textually preceding them in containing scopes to be semantically analysed before they can be processed concurrently with the outer scopes. Since the semantic analysis of statements adds nothing to the symbol tables, this additional wait is needless. The same situation holds if DKY handling is used—resolution of a DKY may be delayed by some statement processing in a parent scope which does nothing to resolve the DKY.

Using two part semantic analysis, a hybrid between DKY avoidance and DKY handling can be employed. For many programming languages, source programs contain more statements than declarations. If declaration processing requires significantly less effort than statement processing, then it may be advantageous to avoid DKYs during statement analysis, while using DKY handling during declaration analysis.

```
PROCEDURE Lookup (identifier, referenceScope)
    searchScope :=referenceScope
    LOOP
        Lock semaphore on symbol table of searchScope.
        LOOP
            Search for a definition of identifier in the symbol table.
            IF found THEN
                IF entry is locked THEN
                    IF referenceScope searchScope THEN
                        Release semaphore and Wait. (* DKY *)
                        RETURN definition.
                    ELSE
                        ERROR. (* Wait would deadlock*)
                    END IF
                ELSE
                    Release semaphore on symbol table of searchScope.
                    RETURN definition.
                END IF
            ELSIF not found and symbol table is incomplete THEN
                IF referenceScope searchScope THEN
                    Release semaphore and Wait. (* DKY *)
                    Lock semaphore on symbol table of searchScope.
                END IF
            ELSE
                EXIT
            END IF
        END LOOP
        Release semaphore on symbol table of searchScope.
        EXIT WHEN searchScope = outermost scope.
        searchScope := parent of searchScope.
    END LOOP
    RETURN NIL. (* No definition found*)
END Lookup
```

Figure 4. Identifier look-up routine for DKY handling

In order to guarantee the absence of DKYs during statement analysis, the semantic analysis of the statements of a scope cannot begin until all declarations in the parent scopes which those statements can legally reference are processed. This scheduling criterion is less restrictive than that of a one part semantic analyser using DKY avoidance. This is because only part of the scope is being subjected to the conservative scheduling strategy of DKY avoidance in two part processing, while the entire scope has to be delayed in one part processing.

Pros and cons

The relative merits of DKY avoidance and DKY handling depend heavily on the cross-usage of identifiers in source programs. If most identifiers referenced in a scope are declared within that scope, then there will be few DKYs, allowing the DKY handling scheme to perform better than the DKY avoidance strategy. However, a heavy usage of identifiers declared in parent scopes implies that the overhead involved in handling DKYs may impose a greater penalty than the restricted concurrency of DKY avoidance.

Two part semantic analysis can be employed to two ends. By processing all declarations first, the symbol table for a scope is completed as soon as possible, thereby narrowing the interval in which DKYs can occur. Look-ups that cause DKYs

slow down compilation and thus reduce the speed-up achievable through concurrent processing. Secondly, DKYs during statement analysis can be avoided if the processing of statements is delayed until the declaration analysis of all parent scopes is complete.

THE IMPLEMENTATIONS

The source language for our prototype compiler was Modula-2+. [5] In addition, the compiler itself was written in Modula-2 +. The compiler was implemented to run on a DEC Firefly workstation, a five processor, shared memory MIMD machine. [6,7]

Three different semantic analysers were written for our compiler to experiment with the ideas presented in the previous section. The first used one part DKY avoidance, the second employed one part DKY handling and the third performed semantic analysis in two parts, with DKY handling during declaration processing and DKY avoidance during statement analysis. For our tests, the compiler on] y processed source code up to the end of the semantic analysis phase and no code was generated.

Revelant features of Modula-2+

A program in Modula-2+ consists of a set of implementation modules, a set of definition modules and a program module. An implementation module contains the code which typically implements an abstract data type used in the program. It exports identifiers through one or more stubs, called definition modules, which contain all the information necessary to semantically analyse references to identifiers exported from its associated implementation module. A definition module contains only declarations and procedure stubs, and no statements. The program module contains the main program, which for our purposes differs from an implementation module only in that it has no accompanying definition module. A compilation unit is a single module residing in a file.

There are three possible types of scopes in a Modula-2+ compilation-definition modules, the implementation or program module and its procedures. There are no unnamed scopes in Modula-2 +. Nested modules are not permitted, but nested procedures are allowed. Definition modules contain no nested scopes.

Implicit inheritance of identifiers declared in outer scopes is permitted for procedures, but inheritance is not defined for modules, since they have no parent scope. Identifiers declared in other modules are made visible in a module scope via the IMPORT statement. Processing of an IMPORT statement requires reading and semantically analysing the definition modules whose names appear in that statement. All identifiers declared in a definition module are also considered to be declared in its accompanying implementation module.

Modula-2+ does not allow syntactic intermixing of declarations and statements. Within a scope, all declarations must precede the statements. Declarations and statements are separated by the reserved word BEGIN.

The WITH statement in Modula-2+ is a construct which allows fields of a record-type variable to be referenced without qualification. To accommodate identifier look-ups inside WITH statements each record declared in the program was allocated a symbol table in which all of its fields are defined. Upon entry to a WITH statement,

the symbol table of the appropriate record is added to the end of the chain of symbol tables through which identifier look-ups from that scope proceeded. In this way, the symbol table for the record specified in the WITH statement will be the first one searched for a definition of an identifier referenced inside the WITH statement. At the end of processing a WITH statement, the record's symbol table is unlinked from the lookup chain. This scheme handles nested WITH statements as well.

Structure of the concurrent compilers

The concurrent compiler is invoked on a file containing either an implementation module or a program module. Figure 1 describes the general structure of our compilers. The first column describes the processing that is performed on each definition module that is imported directly or indirectly by the module being compiled. The second column describes the processing applied to the main module and the third column describes the processing for each procedure scope contained in the module.

Processing implementation and program modules

The lexical analysis and splitter processes transform the main module of the source program, with the lexical analyser converting characters to tokens and the splitter dividing the module into its constituent scopes. Separate instances of the parser are created by the splitter to process the code in each procedure and the module body concurrently. The parser processes call procedures to semantically analyse their scope. Pipelines are implemented between the lexical analyser and the splitter, and between the splitter and the parsers. Thus, compilation of the main module requires one lexical analyser process, one splitter process and one parser process for each scope in the module.

Processing definition modules

The token stream for the main module scope is concurrently directed to the import process. As the import process encounters IMPORT statements, it forks instances of the compiler to process the appropriate definition modules. A table of definition modules which have been compiled or are being compiled is kept so that each definition module is compiled exactly once. This action is one of the earliest taken by the compiler since all IMPORT statements appear at the beginning of the module. In addition, if the main module is an implementation module, then instances of the compiler are forked to process its corresponding definition modules. Compilation of a definition module is similar to that of an implementation or program module. The major difference stems from the fact that definition modules contain no nested scopes. Thus, there is no splitter process, and only one instance of a parser exists per definition module. The import process is used to detect cases of nested importation and start appropriate compiler processing.

Employing and restricting concurrency

Since the compilers were implemented in Modula-2+, the language's concurrency features were used to parallelize the compiler. The Thread module provides routines to create, suspend and unblock lightweight processes. [8] Mutual exclusion is implemented with mutexes, a semaphore-like construct.

Since the Firefly has only five processors, allowing unrestricted forking of processes would result in unproductive parallelism. [4,30] Therefore, concurrency control was implemented by building a process control mechanism above the Thread abstraction. Only a fixed number of processes were allowed to run at any given time. Calls to the process creation routine (Thread. Fork) and the process suspension routine (Thread.Wait) were replaced by calls to similar procedures of the process control module, Scheduler. This module allowed processes to be activated only when the number of currently executing processes was less than the number of processors.

One part DKY avoidance

The mm1a compiler performed one part semantic analysis with DKY avoidance. The compiler began by starting compilation of the main module. Processes were created to compile the definition modules of the main module (if any) and all imported definition modules.

The compilers processing definition modules in turn forked processes to compile their imported modules. Processing of a module scope was suspended until all of its imported definition modules were semantically analysed. When these definition modules were finally compiled, processing of the main module continued, and the declarations in the body of this scope were processed, with processes forked to semantically analyse procedure scopes as they were encountered.

One part DKY handling

The mm1h compiler performed semantic analysis in one part using DKY handling. Compilation of a scope was started as soon as possible. For the main module being compiled, this happened upon invocation of the compiler. Procedure scopes were parsed and semantically analysed as soon as they were detected by the splitter. Compilation of a definition module was started as soon as an import of that module was detected by the import process. Processes compiling modules did not wait for their imported definition modules to be completely processed before continuing.

The completion of the symbol table of a scope was signalled at the end of declaration processing for that scope (i.e. when the reserved word BEGIN was seen). It is at this point that all processes waiting for declarations of undeclared identifiers in the symbol table of that scope were unblocked.

Two part DKY handling

In Modula-2+, the strict ordering of declarations before statements in a scope implies that statement semantic analysis will never delay the resolution of a DKY in one part DKY handling or the forking of a semantic analyser to process a child scope in one part DKY avoidance. Therefore, two part semantic analysis with DKY avoidance is the same as one part semantic analysis with DKY avoidance.

The fact that Modula-2+ requires all of the declarations in a scope to precede all the statements removes one of the major potential advantages of two part semantic analysis as compared to a one part scheme. However, the merits of avoiding DKYs during statement analysis and allowing them during declaration analysis could still be tested. Thus, the only two part semantic analyser which was constructed (compiler mm2h) used DKY handling during declaration analysis and delayed the statement analysis of a scope until all the tables which it could possibly search were completed. The symbol table of a module scope was complete when all of its declarations were processed and the tables of its imported definition modules were complete. For a procedure scope, this event occurred when its tables, its parent procedures' tables and its enclosing module's tables were complete.

The semantic analysis of a scope began as it would have in one part DKY handling. Semantic processing of a scope began as soon as possible, with DKYs allowed to occur. However, when the reserved word BEGIN was encountered in that scope (signifying the end of the declarations), semantic analysis was halted until all the symbol tables which that process could search were completely built. Then, statement analysis could be conducted on the scope without DKYs occurring.

PERFORMANCE RESULTS

In this section we evaluate the performance of the compilers described in the preceding section. Several statistics were gathered, both regarding the compilers and the source programs which they compiled. * The experiments were conducted on a five processor Firefly workstation which was otherwise idle.

The source programs used for these tests were drawn from a large program library, which was made available by the Digital Equipment Corporation Systems Research Center. We gathered statistics on the compilation of 40 different source programs that were written by a diverse set of authors. The statistics presented in this paper are drawn from a set of twelve programs that form a representative subset of our data. More detailed results are available in Reference 31. A general description of these programs is given in Table I. None of these programs had semantic errors. The compilers were instrumented to measure the following statistics:

1. Speed-up over a sequential compiler for one to five processors for all three compilers.
2. Average number of processes which were kept active for all three compilers.
3. Number and types of DKYs which occurred in mm1h and mm2h.

In order to better understand these results, the following data were obtained regarding the source programs that were being compiled:

1. Number and types of scopes in the source programs, along with their nesting depth.
2. Identifier usage between scopes in the source programs.

An existing sequential Modula-2+ compiler (called mp) was used as the baseline for speed-up comparisons.

* A much more detailed presentation of our experimental results is given in Reference 31.

Table I. Description of test suite

Name	No. of lines in main module	Definition modules	Description
Linkage. mod	142	6 files, 577 lines	Manager of lists of symbols and mutexes
TimeConv.mod	354	9 files. 1074 lines	Converter of times from different time zones
Clock. mod	444	14 files, 3611 lines	Software clock manager
ColorCodeServer.mod	1390	14 files, 1359 lines	Server for colour manager
ColorCode.mod	1498	14 files, 1371 lines	Manager of colour states
Symbol.mod	689	26 files, 3180 lines	Compiler table manager
Basic VM.mod	1019	14 files, 3832 lines	Memory manager software
DECnet.mod	1839	19 files, 4557 lines	DECnet network software
M2Macro.mod	1398	11 files, 2334 lines	Preprocessor for Modula-2
NSP.mod	4000	18 files, 4465 lines	Communication software
High TTD. mod	2863	27 files, 5082 lines	Module used by debugger
Disk. mod	2168	19 files, 5504 lines	Driver for MSCP

Source program statistics

Table II gives detailed characteristics of the source programs described in Table I. For each program, the number and type of scopes is given, along with the number of procedure scopes at each nesting depth. Note that each program contains only one implementation/program module—the remaining modules are definition modules.

Also given are the sequential compilation times to the end of semantic analysis, with the breakdown given of the fraction of that time spent in declaration analysis and in statement analysis. These numbers were derived from the mp compiler. The programs are listed in order of increasing size, where size is defined to be sequential compilation time.

Table II. Characterization of test programs

Name	Modules (implementation/ program and definition)	Procedures (nesting depth) 1	2	3	Compile time (sequential),s	Declaration analysis, per cent	Statement analysis, per cent
Linkage.mod	7	5	2	1	5·0	86	14
TimeConv.mod	10	9	0	0	11·0	69	31
Clock.mod	15	15	1	0	19·3	84	16
ColorCodeServer.mod	15	25	0	0	20·0	89	11
ColorCode.mod	15	24	0	0	22·3	87	13
Symbol. mod	27	26	0	0	25·4	69	31
Basic VM.mod	15	48	3	0	29·1	80	20
DECnet.mod	20	18	8	0	32·2	82	18
M2Macro.mod	12	53	27	0	35·1	86	14
NSP.mod	19	27	27	1	44·5	88	12
High TTD. mod	28	149	6	0	57·7	72	28
Disk. mod	20	67	2	0	69·4	53	47

There are three interesting observations that may be made here. First, the smaller programs appear to have roughly equal numbers of module scopes and procedure scopes, whereas the larger ones have more procedure scopes. Thus we may say that the number of procedure scopes in a program grows at a faster rate with program size than the number of module scopes. The second observation is that the fraction of time spent in declaration semantic analysis is approximately 75 per cent of the time spent in semantic analysis, and this ratio appears to be independent of program size. Finally, note that there are very few nested procedures—the vast majority of procedure scopes are at nesting level 1.

Table III gives the statistics on the cross-usage of identifiers in the source programs. The leftmost column identifies the type of reference:

1. *Local—an* identifier declared in the scope in which the reference occurred.
2. *Non-local—* an identifier in an enclosing scope.
3. *Imported—* an identifier imported from a definition module.
4. *Record field—* an identifier that names a field in a record.
5. *Built-in—* the name of a built-in object (e.g. library routines such as sqrt) provided by the compiler.

The top row of this table identifies the source of the reference (i.e. where the reference occurred):

(a) *def module—* a definition module
(b) *main decl* — declarations in the main module
(c) *main stmt* —statements in the main module outside its procedures
(d) *proc decl—* declaration in a procedure
(e) *proc stmt—* statements in a procedure.

There are some important observations which can be made about this data. First, references to locally declared identifiers form the largest single type of reference. Secondly, most non-local references are from procedure statements to identifiers in the main module. The vast majority of references to identifiers arise from statements in procedures. This implies that it is imperative to have the symbol table of the main module complete as early as possible. Finally, the large number of references arising in definition modules is indicative of the use of hierarchical definition of higher level abstractions from lower level abstractions.

The empty table entries for non-locals occurs because a module scope cannot contain ‘non-local’ references, since they have no enclosing scope. The empty table

Table III. Percentage distribution of identifier references

Referent	Source of reference					Total
	def module	main decl	main stmt	proc decl	proc stmt	
Local	24·21	1·19	0·25	0·58	18·56	44·79
Non-local				5·41	16·81	22·20
Imported	4·45	0·54	0·00	1·37	4·17	10·54
Record field			0·00		11·21	11·21
Built-in	5·69	0·27	0·10	1·74	3·48	11·26
Total	34·35	1·99	0·37	9·05	54·28	100·00

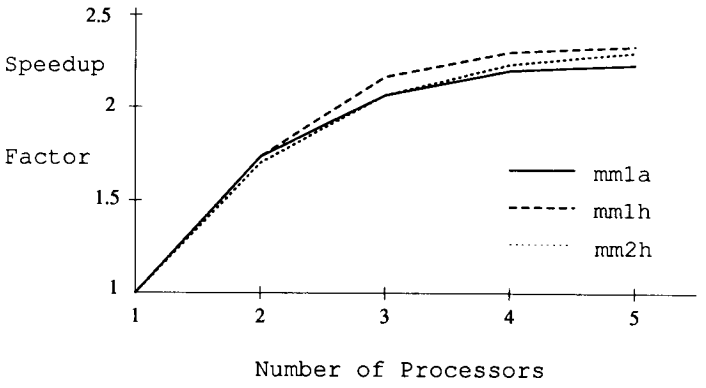

Figure 5. Speed-up curves for concurrent semantic analysis

entries for record fields occur because record fields cannot be referenced in declarations, and are therefore unreferenced in definition modules, and in procedures' and main modules' declarations.

Compiler statistics

The speed-up curves for the compilers (relative to the sequential mp compiler) on the test programs are given in Figure 5. * The speed-up is relatively similar for all compilers, with mm1h showing somewhat better results than mm1a and mm2h. The speed-ups for individual programs ranged from 1·9 to 2·5 on five processors for mm1a and from 2·0 to 2·7 on five processors for mm1h and mm2h. In general, greater speed-up was observed when compiling larger programs. These results are similar to Vandevoorde's C compiler (2·3 to 3·0 on a five-processor Firefly). However, his compiler performed code generation as well, while our compilers do not. We expect our speed-up results to improve when code generation is added.

Table IV shows the average number of active processes during compilation with five processors on the source programs. As expected, mm1a keeps fewer processes active than either of the DKY handling compilers, by about 0·6 processes per compilation.

The symbol table look-up and definition routines were instrumented to gather statistics on the number and types of DKYs which occurred in mm1h and mm2h.

Table IV. Active processes

	mm1a	mm1h	mm2h
Average number of live processes	4·0	4·6	4·5
Average speed-up	2·2	2·3	2·2

* These results are the averages from 10 separate tests. The standard deviation of the results from these runs was usually less than 1 per cent of the elapsed time, and never more than 2 per cent.

Table V. Percentage distribution of DKYs for mm1h

Type of DKY	Source of DKY					Total
	def module	main decl	main stmt	proc decl	proc stmt	
Incomplete						
P-Mod	26·45	1·18	0·00	2·11	3·29	33·02
P-Proc			0·08		4·72	4·80
Local	0·00	0·25	0·00	1·43	0·93	2·61
No entry	12·89	0·25		37·57	8·85	59·56
Total	39·34	1·68	0·08	41·11	17·78	100·00

Table VI. Percentage distribution of DKYs for mm2h

Type of DKY	Source of DKY			Total
	def module	main	proc	
Incomplete				
P-Mod	29·82	1·61	5·79	37·23
P-Proc				
Local	0·57	0·09	0·28	0·95
No entry	13·87	0·19	46·82	61·06
Total	44·25	1·90	51·38	100·00

These results are summarized in Table V, VI and VII. Tables V and VI show the types of DKYs and the frequencies at which they occurred when semantically analysing declarations and statements in each of the three types of scopes for mm1h and mm2h. In these tables, we distinguish three kinds of DKYs arising from incomplete symbol table entries. *Local DKYs* are due to the temporary symbol table locking during the completion of a complex symbol table entry. *P-Proc DKYs* are due to waits for the completion of entries for named procedure like scopes. *P-Mod DKYs* are caused by waits for the completion of entries for named module like scopes.

From these statistics, it can be determined that the majority of DKYs occurred on identifiers that had not yet been entered into the symbol tables. In addition, DKYs on local entries were insignificant. DKYs on placeholder module identifiers (i.e. P-Mod DKYs) were significant, accounting for one third of the DKYs. Finally,

Table VII. Percentage distribution of DKY targets

DKY target	Compiler	
	mm1h	mm2h
def module	40·19	44·54
main decl	52·99	49·57
proc decl	6·57	7·69

the vast majority of DKYs occurred during the processing of declarations, both in procedures and in modules. Note that DKYs on placeholder procedure entries during declaration processing do not occur, since procedures cannot be referenced in declarations.

The distribution of DKY targets (i.e. where the identifiers that were waited upon were declared) is given in Table VII. The high percentage of DKY targets in definition modules is indicative of the hierarchical definition chains that are typical in large software systems.

Finally Table VIII shows, for each source program, the ratio of DKYs to identifier references in the compilation of the program using five processors. As can be seen, fewer than 5 per cent of all identifier references cause DKYs during semantic analysis. Furthermore, the numbers for mm1h and mm2h are very similar in almost all cases.

Discussion

General results

The curves in Figure 5 show that on average, the compilers achieved a speed-up of approximately 2·3. This result is somewhat disappointing—however, there are explanations for this.

First, the compilers did not perform code generation, * which is a major, CPU-intensive phase of compilation. We expect that with code generation, the speed-up will be significantly higher. Secondly, the Firefly is not a totally symmetric machine. In particular, I/O processing is handled by a designated processor, and thus it cannot contribute fully to the parallel processing. This explains the flat part of the speed-up curve between 4 and 5 processors. Finally, contention on the shared memory bus on the Firefly causes speed-up to be significantly less than ideal. Analytic perform-

Table VIII. Ratio of number of DKYs to number of identifier references

Program	mm1h	mm2h
Linkage.mod	0·043	0·043
TimeConv.mod	0·029	0·029
Clock.mod	0·039	0·032
ColorCodeServer.mod	0·019	0·018
ColorCode.mod	0·018	0·018
Symbol.mod	0·033	0·036
Basic VM.mod	0·036	0·037
DECnet.mod	0·021	0·027
M2Macro.mod	0·029	0·015
NSP.mod	0·020	0·019
HighTTD.mod	0·032	0·022
Disk. mod	0·032	0·028

* The code generation phase of the base compiler we were using had not been modified for concurrent processing at the time the research reported in this paper was performed.

ante measures [6] show that the degradations in adding processors to the Firefly's shared bus are 0·89, 0·87, 0·86 and 0·84 for two, three, four and five processors. This implies that even if one more compiler process can be kept busy, the bus contention will cause the speed-up to be less than ideal.

DKY avoidance vs DKY handling

In comparing the DKY avoidance and DKY handling strategies, we must look at the results for mm1a and mm1h. From our results, it is obvious that mm1h only performs marginally better (between 5 and 10 per cent better) than mm1a. We had expected DKY handling to outperform DKY avoidance, but by a larger margin. The results obtained can be explained by several factors.

First, DKY handling breaks down when there is a large degree of identifier cross-usage between scopes of the source program. From the source program statistics in Table III, we see that a large portion of the identifiers referenced in a scope are either imported or declared in an enclosing scope (approximately 33 per cent). Secondly, scopes are not nested very deeply in the source programs—nearly all procedure scopes are at nesting level one. The DKY avoidance strategy performs badly when there is a deep nesting of scopes, since the semantic analysis of these scopes must be delayed until enclosing scopes are processed. Since there is not very much deep nesting, DKY handling and DKY avoidance will perform similarly.

Finally, the traditional approach to handling built-in identifiers is for the compiler to put them in a virtual scope which conceptually encloses the main program, and to provide a symbol table for this virtual scope. This is often done for implementor convenience, even in languages like Modula-2 + where redeclaration of built-in identifiers is prohibited. This is a poor strategy for parallel compilers that use DKY handling, since it causes an unnecessary DKY on the main module scope. When a reference to a built-in identifier must search through the main module scope before reaching the enclosing virtual scope. This effect is not negligible, since in the test programs used, built-in identifiers comprise 11 per cent of the identifiers referenced. In retrospect, we should have used a different strategy for handling built-in identifiers when using DKY handling (e.g. including the built-in identifiers in each scope).

It is also interesting to note that although mm1h managed to keep approximately 0·6 more processes active than mm1a, the difference in speed-up is only 0·1. This leads us to believe that the processes in mm1h are performing extra, unproductive work. Some of this can be explained by the fact that there is additional contention for the locks on the symbol tables (especially for the symbol table of the main module) in mm1h. The additional contention in mm1h is explained by the fact that when processes are awakened, they immediately try to reacquire the lock on the symbol table that they were holding when they blocked. This lock is still being held by the signalling process. This situation does not arise in mm1a. Using a system provided utility on the Firefly, contention on locks can be measured. * The extra time spent by processes in mm1h waiting for the lock on the symbol tables to be released accounted for approximately 60 per cent of the extra processing done by mm1h processes. Almost all of this overhead was caused by the lock on the symbol table of the main module.

* The locks are implemented as queues of waiting processes so a processor does not spin waiting on a lock.

One part vs two part processing

In comparing one part semantic analysis with the two part semantic analysis strategy, the results obtained for mm1h and mm2h are examined. From the speed-up curves, mm1h seems to perform slightly better than mm2h. This can be explained by three factors.

First, from Table II, it is clear that approximately 75 per cent of the effort (i.e. processing time) in semantic analysis is spent in declaration processing. Since DKY handling is used in declaration processing for both mm1h and mm2h, it would be expected that the two compilers performed very similarly. mm2h outperformed mm1h by 1 to 3 per cent on the source modules TimeConv.mod, Clock. mod and Disk.mod. Secondly, Tables V and VI show that only a small fraction (approximately 18 per cent) of the DKYs in mm1h occurred during statement semantic analysis, and therefore DKY avoidance during this phase would be inferior to DKY handling. Finally, as stated above, Modula-2+ does not allow syntactic intermixing of declarations and statements, thus removing one possible benefit of two part semantic analysis. Even for a language with intermixing it is not clear how great this benefit would have been since statement processing requires only a small portion of the semantic analysis time.

CONCLUSIONS

We have presented the results of an investigation into how the semantic analyser of a concurrent compiler should be structured in order to maximize performance. The major problem that arises is what we call the DKY problem, in which a process analyzing one scope looks into an incomplete symbol table of a containing scope for identifier attribute information. Three solutions were presented: DKY avoidance, DKY handling and two part semantic analysis.

All three approaches performed similarly, achieving an average speed-up of approximately 2·5 on a wide variety of source programs. DKY handling performed about 5 to 10 per cent better than DKY avoidance. The difference was small due to a significant degree of identifier cross-usage between scopes and because of the shallow nesting depth of most of the program scopes. Two part semantic analysis did not outperform one part semantic analysis since statement processing in Modula-2+ takes approximately one-third of the time that declaration processing does. Furthermore, few DKYs were found to occur during statement semantic analysis during one part processing.

AKNOWLEDGEMENTS

The research described in the paper was generously supported by Digital Equipment Corporation of Canada and by the Digital Systems Research Center. V. Seshadri was supported an Ontario Graduate Fellowship.

REFERENCES

1. A. V. Aho, R. Sethi and J. D. Unman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, Massachusetts, 1986.
2. V. Seshadri, I. S. Small and D. B. Wortman, 'Concurrent compilation', *Proceedings of the IFfP WG1O.3 Working Conference on Distributed Processing,* 1987, pp. 627–641.
3. V. Seshadri, D. B. Wortman, M. D. Junkin, S. Weber, C. P. Yu and I. Small, 'Semantic analysis

in a concurrent compiler', *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988, pp. 233–240.
4. M. D. Junkin and D. B. Wortman, 'The implementation of a concurrent compiler', *Technical Report CSRI-235*, Computer Systems Research Institute, University of Toronto, 1990.
5. P. Rovner, R. Levin and J. Wick, 'On extending Modula-2 for building large, integrated systems', *Technical Report 3*, Digital Equipment Corporation Systems Research Center, 1985.
6. C. P. Thacker and L. C. Stewart, 'Firefly: a multiprocessor workstation', *Proceedings of the Second International Conference on Architectural Support for Programming Language and Operating Systems*, 1987, pp. 164–172.
7. C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, 'Firefly: a multiprocessor workstation', *IEEE Trans. Computers*, **C-37,** (8), 909–920 (1988).
8. A. D. Birrell, J. V. Guttag, J. J. Horning and R. Levin, 'Synchronization primitives for a multiprocessor: a formal specification', *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, 1987, pp. 94–102.
9. P. R. McJones and G. F. Swart, 'Evolving the UNIX system interface to support multithreaded programs', *Technical Report 21*, Digital Equipment Corporation Systems Research Center, 1987.
10. B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation and Implementation*, Holt, Rinehart and Winston, New York, New York, 1983.
11. C. N. Fischer, 'On parsing context free languages in parallel environments', *Ph.D. Thesis*, Cornell University, 1975.
12. J. Cohen, T. Hickey and J. Katcoff, 'Upper bounds for speedup in parallel parsing', *Journal of the ACM*, **29,** (2), 408–428 (1982).
13. J. Cohen and S. Kolodner, 'Estimating the speedup in parallel parsing', *IEEE Trans. Software Engineering*, **SE-11,** (l), 114–124 (1985).
14. R. op den Akker, H. Alblas, A. Nijholt and P. Oude Luttighuis, 'An annotated bibliography on parallel parsing', *Technical Report Memoranda Informatica 89-67*, Department of Computer Science, University of Twente, 1989.
15. G. V. Cormack, 'An LR substring parser for noncorrecting syntax error recovery', *ACM SIGPLAN Notices*, **24,** (7), 166–169 (1989).
16. R. M. Schell, 'Methods for constructing parallel compilers for use in a multiprocessor environment', *Ph. D. Thesis*, University of Illinois at Urbana-Champaign, 1979.
17. M. K. Donegan and S. W. Katzke, 'Lexical analysis and parsing techniques for a vector machine', *Proceedings of the ACM Conference on Programming Languages and Compilers for Parallel and Vector Machines, 1975*, pp. 138-145.
18. C. A. Ellis, 'Parallel compiling techniques', *Proceedings of the ACM National Conference*, 1971, pp. 508–519.
19. H. E. Krohn, 'A parallel approach to code generation for FORTRAN-like compilers', *Proceedings of the A CM Conference on Programming Language and Compilers for Parallel and Vector Machines*, 1975, pp. 146–152.
20. N. Lincoln, 'Parallel compiling techniques for compilers', *ACM SIGPLAN Notices*, **5** (10), 18–31 (1970).
21. M. Zosel, 'A parallel approach to compilation', *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1973, pp. 59–70.
22. J-L. Baer and C. S. Ellis, 'Model, design, and evaluation of a compiler for a parallel processing environment', *IEEE Trans. Software Engineering*, **SE-3** (6), 394–405 (1977).
23. T. Christopher, O. E1-Dessouki, M. Evens, H. Harr, H. Klawans, P. Krystosek, R. Mirchandani and Y. Tarhan, 'SALAD: a distributed compiler for distributed systems', *Proceedings of the International Conference on Parallel Processing, 1981*, pp. 50–57.
24. W. Huen, O. E1-Dessouki, E. Huske and M. Evens, 'A pipelined DYNAMO compiler', *Proceedings of the International Conference on Parallel Processing, 1977*, pp. *57–66.*
25. J. A. Miller and R. J. LeBlanc, 'Distributed compilation: a case study', *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 1982, pp. 548–553.
26. J. L. Frankel, 'The architecture of closely-coupled distributed computers and their language processors', *Ph. D. Thesis*, Harvard University, 1983.
27. M. T. Vandevoorde, 'Parallel compilation on a tightly-coupled multiprocessor', *Technical Report 26*, Digital Equipment Corporation Systems Research Center, 1988.
28. H-J. Boehm and W. Zwaenepoel, 'Parallel attribute grammar evaluation', *Technical Report COMP TR87-55*, Department of Computer Science, Rice University, 1987.

29. R. C. Holt and J. Hume, *Introduction to Computer Science Using the Turing Programming Language*, Reston Publishing Company, Inc., Reston, Virginia, 1984.
30. M. T. Vandevoorde and E. Roberts, 'Workcrews: an abstraction for controlling parallelism', *Technical Report 42*, Digital Equipment Corporation Systems Research Center, 1989.
31. V. Seshadri, 'Concurrent semantic analysis', *Master's Thesis*, Department of Electrical Engineering, University of Toronto, 1988. Reprinted as *Computer Systems Research Institute Technical Report CSRI-216.*