Semantic Analysis in a Concurrent Compiler

V. Seshadri, D.B. Wortman, M.D. Junkin
S. Weber, C.P. Yu, I. Small

Computer Systems Research Institute
University of Toronto
Toronto, Ontario, Canada

Abstract

Traditional compilers are usually sequential programs that serially process source programs through lexical analysis, syntax analysis, semantic analysis and code generation. The availability of multiprocessor computers has made it feasible to consider alternatives to this serial compilation process. The authors are currently engaged in a project to devise ways of structuring compilers so that they can take advantage of modern multiprocessor hardware. This paper is about the most difficult aspect of concurrent compilation: concurrent semantic analysis.

1. Introduction

We begin by describing the structure of our concurrent compiler. Section 3 describes the compiler table management issues that arise during concurrent semantic analysis. It includes a novel algorithm for optimizing table search. In Section 4 we describe the *Doesn't Know Yet* problem which arises when one process searches a compiler table that is being concurrently constructed by some other process. We describe several strategies for dealing with this problem.

1.1 Compiler Model

The canonical sequential compiler [1,22] is shown in Figure 1. It is organized into a number of phases each of which accepts an input stream from the previous phase and produces an output stream for the following phase. Lexical analysis converts a stream of characters into a stream of language-specific lexical tokens. Syntax analysis converts the stream of lexical tokens into a data structure (parse tree or equivalent linear stream) that describes the structure of the source program. The semantic analysis phase performs context checking to verify that the source program obeys all the non-syntactic constraints imposed by the programming language. Semantic analysis processes declarations and records information from these declarations in symbol and type tables that are used by subsequent phases. The storage allocation phase assigns addresses to variables and lays out the run-time storage for procedures and functions. The optimization and code generation phases transform the source program into instructions for the target computer.

An important characteristic of the sequential compiler is that the processing has been designed to work well for programming languages which have nested scopes of declaration and require declaration before use. The natural processing order of the sequential compiler is exactly the right one for such languages. A hidden invariant in such compilers is that the compiler tables for outer scopes are completely built before inner scopes are processed. This invariant is false in the concurrent compilers that we are considering.

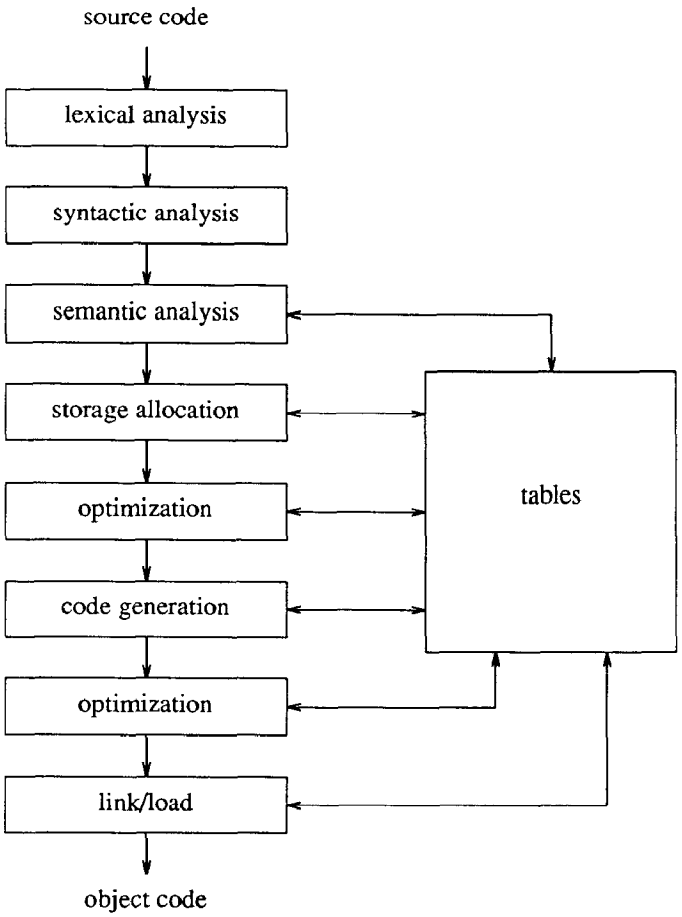

Figure 1: Canonical Compiler Structure

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

2. Concurrent Compiler Structure

2.1 Concurrent Compiler Development Project

The Concurrent Compiler Development Project at the University of Toronto is an effort to aggressively utilize multiprocessing hardware in order to accelerate compilation. An overview of the project appears in [4]. Initially, we are concentrating on well-behaved source languages which:

- are block structured
- follow scope rules similar to those of Algol-60
- require identifiers to be declared before they are used
- can allow declarations to be syntactically intermixed with statements
- have reserved words, rather than keywords, that determine program structure.

Modern, block structured languages such as Pascal, C, Modula-2 and Ada in general satisfy our criteria. Although less disciplined languages such as Fortran and PL/I do not meet some of these requirements, our compilation techniques can, with some adaptation, be applied to them as well. To evaluate our ideas on concurrent compilation we are presently building a concurrent compiler for Modula-2 to run on an MIMD multiprocessor [17].

The approach we are following is based on three general principles.

- *redistribution of effort.* We are diverging from traditional compiler structure by moving some tasks from one compiler phase to another in cases where doing so will allow us to process source programs more concurrently. For example, we have moved a small amount of rudimentary parsing functionality into the lexical analyzer so that it can recognize major structural boundaries in the source program.
- *sensible subdivision.* Unlike some of the early work on concurrent compilation, we are dividing the program being processed only at boundaries that make sense in terms of the compiler processing that needs to be performed. For example, our lexical analyzer deals with include files concurrently. We divide the source program at the scope (module, procedure, function) level because doing so yields convenient segments for further processing.
- *accelerated initiation.* We attempt to start processing program segments at the earliest possible time. In section 4 of this paper we discuss several variations on this strategy in the context of semantic analysis.

2.2 Previous Work

The earliest attempts at concurrent compilation date back to the early 1970's ([8], [9], [12], [13], [19]). Most of these efforts were restricted to making a specific phase of compilation concurrent for a specific language (i.e., FORTRAN, APL) on vector or array processors. Other work on concurrent compilation ([2], [5], [11], [15], [21]) has dealt with the use of pipelining to speed compilation. More recent work by Lipkie [14], Schell [16] and Vandevoorde [18] is more general, dealing with modern, block structured languages and architectures of the MIMD model. These works are more relevant to the material presented here.

Most of the theoretical work on concurrent compilation has concentrated on parsing. Fischer [10] was the first to extensively study concurrent parsing. His thesis dealt primarily with proving the correctness of various concurrent parsing schemes. Cohen et al. ([6], [7]) extended Fischer's work by attempting to determine the speedup offered by concurrent parsing. The theoretical work on concurrent parsing generally assumed that the token stream for the source program was arbitrarily divided into some number of segments. A complicated algorithm was then required to merge the states of adjacent parsers when they ran out of tokens.

2.3 Making Compilation Concurrent

The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors. This approach is limited in two ways. First, the degree of concurrency is limited by the number of stages in the pipeline, which results in the utilization of only a fixed number of processors. Second, the overall speed of compilation is constrained by the speed of the slowest pipeline stage.

A more effective way to utilize multiprocessing resources is to split the source program into segments which are then compiled concurrently. This is the approach taken in [14], [16] and [18], and forms the basis of the model presented here. In our concurrent compiler, a sequential lexical analyzer splits the source code into many segments, which are concurrently processed through the various phases of compilation before a final, merging pass recombines the object code produced into a single program. At any given point in time, there exist many compiler processes, each performing some phase of compilation on some program segment. A diagram of our compiler structure is given in Appendix A.

To avoid introducing unnecessary processing, the source program is not split into arbitrary segments. The lexical analyzer breaks the code into segments which correspond to the source language's scope constructs. Typically, these are balanced structures such as procedure and function bodies and modules, which can be easily recognized and separated during lexical analysis. The lexical analyzer constructs a tree out of these segments, reflecting the nesting structure of the program. Thus, many token streams are created, with nested scopes in a stream replaced by pointers to other token streams (Figure 2). The subsequent phases of the concurrent compiler then operate on individual nodes of this tree.

Concurrent parsing schemes such as those suggested in [6], [7] and [10] assume that the source program is divided into arbitrary segments. One result of this assumption is that the location of syntax errors is difficult to pinpoint correctly. By splitting the program into pieces corresponding to major syntactic constructs, we hope to achieve good error recovery without significant overhead. We also avoid the high overhead reported by Cohen [7] of merging parsers operating on arbitrary program segments. We do have to guarantee that the splitting performed by the lexical analyzer neither introduces nor obscures syntax errors.

Scope-level code division also leads to a reduction in the communication required between processes in the various phases of compilation. Syntactic analysis of a code segment can be performed independently of the parsing of any other code segment. Storage can be allocated for variables declared in a scope independently of other scopes. Code generation can be performed concurrently at the procedure and function body level, with the object code segments merged using traditional linking technology.

This reduction in communication overhead does not extend to semantic analysis. Performing semantic analysis concurrently at the scope level of granularity requires extensive interprocess communication due to the flow of semantic information between scopes. Identifiers declared in one scope may be legally referenced in another, subject to the specific visibility rules of the language. Since the process which is semantically analyzing the scope of reference is not necessarily the same process which is processing the scope of declaration, information must be exchanged between the two processes.

3. Table Management for Concurrent Compilation

In traditional procedural languages (e.g., Pascal, Modula-2) a program consists of some number of *scopes* (e.g., begin-end blocks, the bodies of procedures and functions) where new identifiers can be declared. There is a concept of *scope parentage*; the main program is the ultimate parent and all other scopes are children of the scopes in which they are nested. We assume that scope parentage is a static property of the program text that can be mechanically inferred. For example, in Figure 2, module A is the parent scope of procedures B and D. Function C has procedure B as its parent and module A as its grandparent. In the languages we are considering there is a *name inheritance mechanism* which controls the visibility of names in nested scopes. The mechanism may be automatic inheritance as in Pascal or controlled inheritance via explicit imports as in Modula-2 modules. The important point for this discussion is that a child scope may make reference to names declared in its parents' scopes. In sequential compilers scopes are naturally processed from outermost to innermost (i.e., parents before children).

3.1 Table Management Issues

We assume that information about declared identifiers is stored in a symbol table [1,20,22]. We also assume that information about built-in and declared types is stored in a separate type table and that there is a scope table which is used to record information about each scope including pointers to the symbol and type tables for the scope. These tables are only written during declaration processing and storage allocation. Semantic analysis can be viewed as being composed of two major sub-phases:

- *Declaration Processing.* In this sub-phase the declarations present in the scope are analyzed. The main function of this phase is to validate declarations, enter newly declared identifiers into the symbol table, enter new types into the type table, and create the necessary links between these tables.

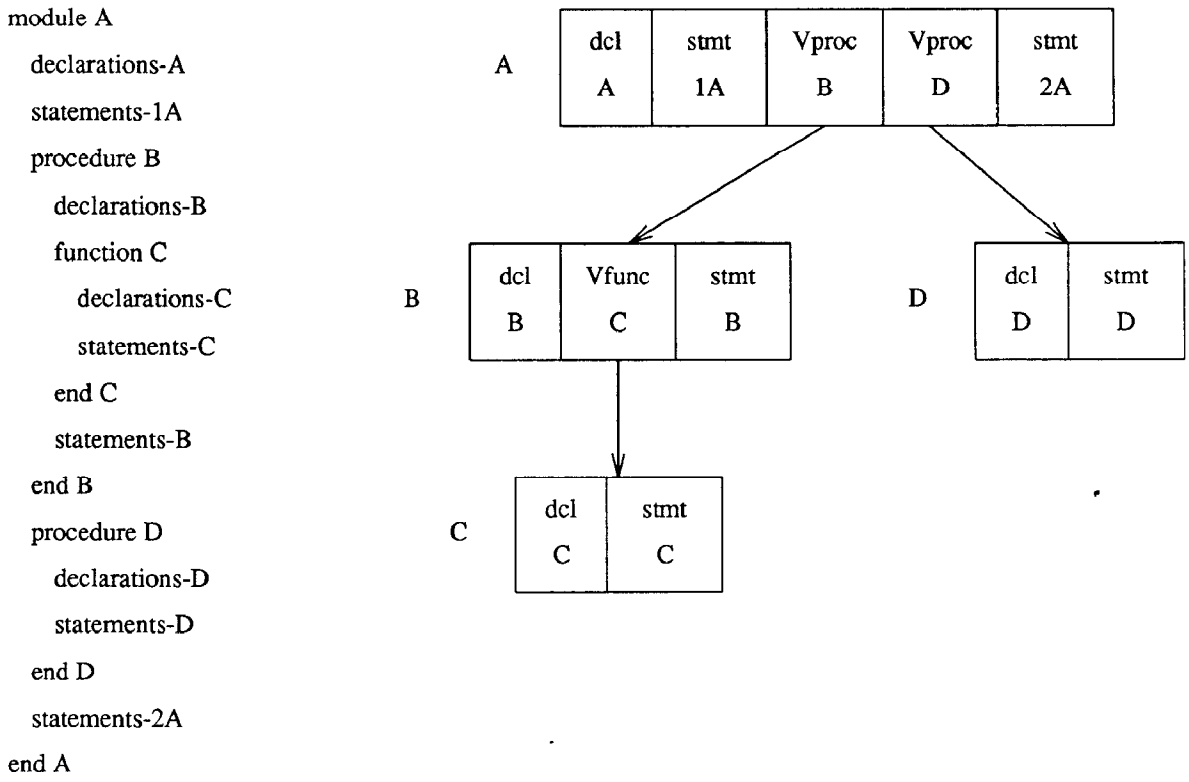

Figure 2: Sample Code Skeleton and Program Tree

- *Statement Processing*. In this sub-phase the executable statements in the scope are analyzed. The main function of this phase is the validation of the semantic constraints imposed by the programming language. We assume that this sub-phase reads the compiler tables but does not modify them.

Concurrent compilation places several constraints on the management of the compiler's tables that are not present in a sequential compiler.

- Access to the tables must be protected by mutual exclusion mechanisms which prevent the tables from being corrupted by overlapping writes. Reads from the tables must behave atomically (i.e., each read must return a complete and consistent table entry).
- Symbol table search mechanisms must be designed to deal with tables that are being modified asynchronously.
- Symbol table search mechanisms must be designed to deal with tables that are incomplete (see Section 4).

The choice of solutions to these problems will have a significant effect on compiler performance. Our goal is to find solutions that will maximize the amount of concurrent table access that can be performed while simultaneously guaranteeing correctness of the symbol table mechanisms. Solutions that cause deadlocks are unacceptable. Solutions that cause excessive delay or redundant processing should be avoided. Because the compiler tables are the means by which concurrently executing compiler phases communicate, the amount of time during which the table is being used exclusively by one process must be minimized.

There are several possible solutions for providing mutually exclusive access to the compiler tables. We describe several of these solutions for one table with the understanding that they can be applied to all tables.

3.1.1 Global Table Locking

One solution is to associate a semaphore with the entire table and use it to lock the table before performing an addition or modification. This solution is very simple to implement but it provides very poor concurrent performance. Any process that wants to write the table may be delayed for a considerable period of time waiting for reads in progress to complete.

3.1.2 Scope Level Locking

A more attractive solution in our model for a concurrent compiler is to divide the tables at scope boundaries and to provide a lock for each scope. This moves the contention between readers and writers from the global level to the scope level and allows the table for other scopes to be searched while the table for one scope is being modified.

3.1.3 Separate Scope and Entry Locking

An optimization of scope level locking is to provide a scope level lock to control additions to the table and table entry level locks to control access to each table entry. The scope level lock would be used when a new entry is being added to the table but only the entry lock would be used when an individual table entry is being modified. The table search mechanism is then designed so that it will wait to process a scope if the scope level lock is set. It must also deal with entry level locks either by waiting for the lock to clear or by skipping around locked entries and retrying them later.

3.2 Optimizing Symbol Table Search

We have developed a strategy for optimizing the symbol table search that avoids unnecessary table searches. This strategy involves developing information about where identifiers are declared earlier in the compilation process than is normally done and then using this information to speed up the search process.

As is done in many compilers, our lexical analysis pass creates an *identifier table* which contains (uniquely) each identifier used in the program. Each identifier occurring in the program text is replaced with an *identifier token* which contains an index for the identifier in the identifier table. This technique is used to reduce the bulk of the program text being processed and to simplify later compiler processing. Usually the identifier table contains only the text for the identifier. We augment each entry with a bit vector containing one bit per scope in the source program. This bit vector is used to record the occurrence of a declaration for the identifier in the scopes. We modify the syntax analysis phase to keep track of current scope and to recognize the declaration of names. Whenever a declaration involving some identifier is recognized, the bit corresponding to the scope of the declaration is turned on in the identifier's bit vector in the identifier table.

When the semantic analysis phase encounters an identifier in some scope it now has two pieces of information that it can use to efficiently locate the scope in which the visible declaration of the identifier is located. The scope parentage of the scope being processed is a list of possible scopes where the identifier *could have been declared.* The bit-vector associated with the identifier in the identifier table is a list of scopes where the identifier *is declared.* By intersecting these two lists we can efficiently discover the relevant scope in which the identifier's declaration occurs or the absence of a visible declaration for the identifier. With this strategy we never need to search more than the symbol table for a single scope. We also avoid the expensive global search usually required to discover that an identifier has not been declared. The disadvantage of this strategy is that syntactic analysis of all of a scope's parent's declarations must be completed before semantic analysis of the scope begins in order to guarantee that the bit vectors are complete.

4. Table Search and the Doesn't Know Yet (DKY) Problem

When a semantic analysis process encounters an identifier referenced in the scope which it is analyzing, it must first search for a declaration of the identifier in the scope. This necessitates a search through the symbol table for that scope. If the desired information is found there processing can continue. However, if it is not found, the process must look for information regarding this identifier in the symbol tables of other scopes in its scope parentage. If semantic analysis of the other scopes is not yet complete when this process searches the tables of those scopes, then it is possible that a visible declaration of the identifier exists in one of those scopes, but has not yet been processed and entered into the appropriate tables. In this case, the process *doesn't know yet* (DKY) whether or not a valid declaration of the identifier exists in the incomplete scope. This is what we term the DKY problem.

The DKY problem creates a bottleneck to the speedup attainable by performing semantic analysis concurrently. This bottleneck manifests itself in three ways:

- The semantic attributes of the identifier in question are not known until the declaration in the appropriate scope has been processed.
- It is not immediately known which scope contains the appropriate declaration.
- In the event that no visible declaration of the identifier exists, all scopes which may contain a visible declaration of the identifier must be processed before this is known and an error message is issued.

We view a solution to the DKY problem as a critical issue for achieving good concurrent compiler performance. It is important to minimize delays introduced by DKYs and to guarantee that all DKY situations are eventually resolved. Although the amount of table searching can be reduced considerably by the strategy described in Section 3.2, the strategy does not completely eliminate the DKY problem. We may know that the identifier we are searching for is declared in a particular scope, but it may not yet have been entered in the tables for that scope.

We have developed two complementary strategies for dealing with the DKY problem. The first strategy is to either control the processing of scopes so that DKYs do not occur (DKY avoidance) or to design the semantic processing algorithms so that DKYs which occur can be dealt with on the fly (DKY handling). The second strategy is to control subdivision of the program so as to minimize DKY related delays. One possibility is to process the declarations and statements in a scope serially (1-part processing). This guarantees that declarations are processed before statements which refer to them but it may delay processing of other scopes. The other possibility is to process the declarations and statements in a scope separately (2-part processing). One advantage of 2-part processing is that DKYs do not occur during statement processing. It also allows storage allocation to proceed concurrently with the processing of statements in the scope. We discuss these strategies in more detail below. A more detailed presentation of this material will appear in [23].

4.1 DKY Avoidance

One solution to the DKY problem is to schedule semantic analysis of scopes so that DKYs can never arise. Scopes are processed from the outermost to the innermost with the scheduling constraint that a scope at nesting level N is not processed until all of its parent scopes at levels less than N have been processed. There is a restriction that before a child scope can be processed concurrently with its parent, any information exported from the child must be processed. Therefore a typical procedure's parameter list must be subjected to semantic analysis before the body of the procedure can be processed concurrently with its parent. Similarly the attributes of names exported from nested modules must be known before the module body can be processed concurrently with its parent. Figures 3 and 4 illustrate scheduling for DKY avoidance with 1 and 2 part processing. For DKY avoidance we assume that processing of statements-X is not started until processing of the corresponding declarations-X has been completed. Although this strategy allows sibling scopes (e.g., B and D) to be processed concurrently, it does not achieve as much concurrency as is possible.

4.2 DKY Handling

An alternative solution is to increase the amount of concurrent processing but provide a mechanism to handle DKYs dynamically when they occur [3]. DKYs arise when the search through a scope parentage chain encounters a scope in which declarations have not been completely processed. The identifier being searched for *may* exist in this scope but this cannot yet be determined. For the search to behave correctly, it must wait for processing of this scope to be completed before it can search further along the parentage chain. If we use the search optimization described in Section 3.2 then at most one scope will be searched and DKYs arise only if the identifier being searched for has not yet been processed in its scope of declaration.

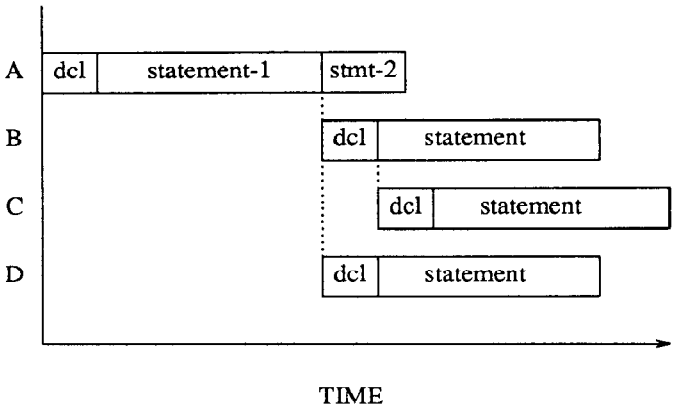

Figure 3: DKY Avoidance with 1-Part Processing

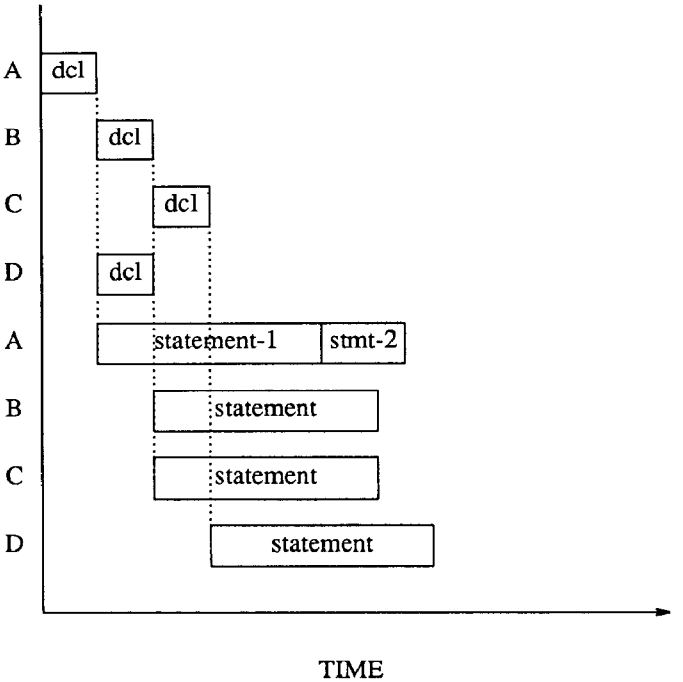

Figure 4: DKY Avoidance with 2-Part Processing

We have identified several possibilities for handling individual DKYs.

- The search process can simply wait for the scope in question to be completely processed and then resume searching.
- Processing of the statement or declaration that caused the search can be suspended, but processing of other statements or declarations can proceed. The partially processed statement is queued for later processing.
- The search process can insert a dummy entry in the table for the incomplete scope. The searching process then either waits for the desired symbol table entry to become available or it continues processing other statements and declarations. Any other process searching for the same table entry will also wait or switch to alternate processing if it encounters a dummy entry. When the process doing semantic analysis encounters this dummy entry when attempting to insert the desired identifier it will first replace the dummy entry with a real entry and then signal the searching process that the desired entry is now available.

Figure 5 illustrates DKY handling with 2 part processing. The difference between Figures 4 and 5 is that DKY handling allows declaration processing to start earlier.

5. Analysis and Conclusions

The proper choice of a strategy for dealing with DKYs depends to some extent on the amount of cross usage of identifiers declared in different scopes. If child scopes make very little use of identifiers declared in their parents then the best way to maximize concurrency is to process as many scopes as possible concurrently and have an efficient way of handling those DKYs which arise. If child scopes tend to make heavy use of identifiers declared in outer scopes then many DKYs will arise and the benefit derived from processing parent and child scopes concurrently would be overwhelmed by the overhead induced by DKY handling. The amount of cross usage is dependent on the programming style of the individuals writing the program. We are gathering statistics on the average characteristics of programs that will let us realistically evaluate the alternative strategies for dealing with DKYs.

It is fairly evident that 2-part processing is superior to 1-part processing. This is because 1-part processing may cause the declaration processing in child scopes to wait for statement processing in the parent to complete when it is not necessary (see Figure 3). This is especially the case when the source program contains nested modules or when the programming language allows declarations and statements to be intermixed (see Figure 4). Based on some preliminary analysis, we are adopting a mixed strategy to cope with the DKY problem. We are using DKY handling at the scope level during declaration analysis while delaying the analysis of statements in a scope until all the relevant declarations have been processed.

Acknowledgement

The research described in this paper has been supported by the Digital Equipment Corporation though its External Research Program and by the Ontario University Research Incentive Fund.

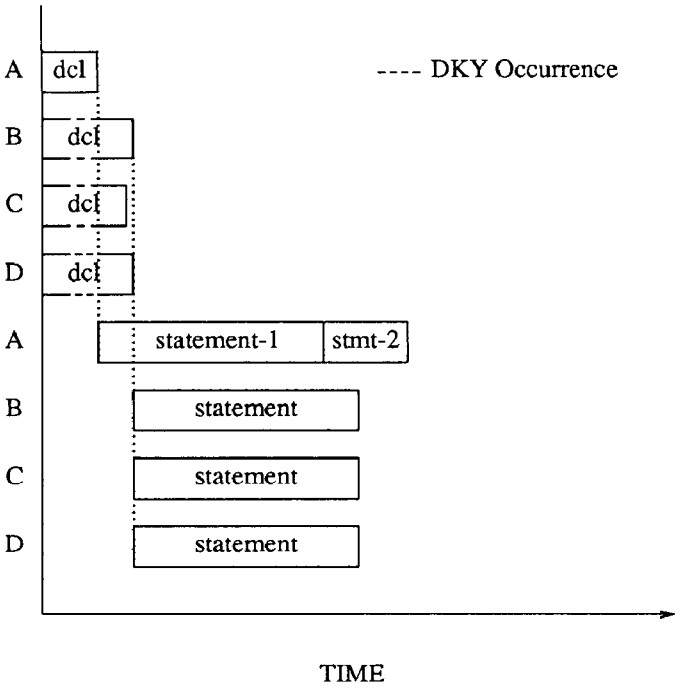

Figure 5: DKY Handling with 2-Part Processing

References

[1] Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools,* Reading, Massachusetts: Addison-Wesley, 1986.

[2] Baer, J-L. and C.S. Ellis, "Model, Design and Evaluation of a Compiler for a Parallel Processing Environment," *IEEE Transactions on Software Engineering,* vol. 3, no. 6, pp. 394-405, November 1977.

[3] Banatre, J.P., J.P. Routeau, and L. Trilling, "An Event Driven Compiling Technique," *Communications of the ACM,* vol. 22, no. 1, pp. 64-75, January 1979.

[4] Seshadri V., I.S. Small and D.B. Wortman, "Concurrent Compilation", Proceedings of the IFIP Conference on Distributed Processing, Amsterdam, Oct 1987.

[5] Christopher, T., O. El-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,* pp. 50-57, August 1981.

[6] Cohen, J., T. Hickey, and J. Katcoff, "Upper Bounds for Speedup in Parallel Parsing," *Journal of the ACM,* pp. 408-428, April 1982.

[7] Cohen, J. and S. Kolodner, "Estimating the Speedup in Parallel Parsing," *IEEE Transactions on Software Engineering,* vol. 11, no. 1, pp. 114-124, January 1985.

[8] Donegan, M.K. 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,"* vol. 10, no. 3, pp. 138-145, Sigplan Notices, March 1975.

[9] Ellis, C.A., "Parallel Compiling Techniques," *Proceedings of the ACM National Conference,* pp. 508-519, August 1971.

[10] Fischer, C.N., "On Parsing Context Free Languages in Parallel Environments," *Ph.D. thesis, Cornell University,* 1975.

[11] Huen, W., O. El-Dessouki, E. Huske, and M. Evens, "A Pipelined DYNAMO Compiler," *Proceedings of the International Conference on Parallel Processing,* pp. 57-66, August 1977.

[12] Krohn, H.E., "A Parallel Approach to Code Generation for Fortran Like Compilers," *Proceedings of the ACM Conference on Programming Languages and Compilers for Parallel and Vector Machines,* vol. 10, no. 3, pp. 146-152, Sigplan Notices, March 1975.

[13] Lincoln, N., "Parallel Compiling Techniques for Compilers," *ACM Sigplan Notices,* vol. 5, no. 10, pp. 18-31, October 1970.

[14] Lipkie, D.E., "A Compiler Design for Multiple Independent Processor Computers," *Ph.D. thesis, University of Washington,* 1979.

[15] Miller, J.A. and J.J. LeBlanc, "Distributed Compilation: A Case Study," *Proceedings of the 3rd International Conference on Distributed Computing Systems (IEEE),* pp. 548-553, October 1982.

[16] Schell, R.M., "Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment," *Ph.D. thesis, University of Illinois,* 1979.

[17] Thacker, C.P. and L.C. Stewart, "Firefly: A Multiprocessor Workstation," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pp.164-172, October 1987.

[18] Vandevoorde, M.T., "Parallel Compilation on a Tightly-Coupled Multiprocessor," *M.Sc. thesis, Massachusetts Institute of Technology,* 1987.

[19] Zosel, M., "A Parallel Approach to Compilation," *Conference Record of the ACM Symposium on Principles of Programming Languages,* pp. 59-70, October 1973.

[20] Graham, S.L., W.N. Joy and O. Roubine, "Hashed Symbol Tables for Languages with Explicit Scope Control", *Proceedings of the Sigplan Symposium on Compiler Construction,* pp. 50-57, August 1979

[21] Frankel, J.L., "The Architecture of Closely-Coupled Distributed Computers and Their Lanugage Processors", *Ph.D. dissertation, Dept of Applied Mathematics, Harvard University,* May 1986.

[22] Fischer C.N. and R.J. LeBlanc Jr., *Crafting a Compiler,* Menlo Park, Calif., Benjamin/Cummings Publishing Co., 1988.

[23] V. Seshadri, "On Concurrent Semantic Analysis", *M.A.Sc. thesis, Dept. of Electrical Engineering, University of Toronto,* expected 1988

Appendix A: Structure of the Concurrent Compiler