A Concurrent Compiler for Modula-2+

David B. Wortman,
Michael D. Junkin*
Computer Systems Research Institute
University of Toronto
6 Kings College Road
Toronto, Ontario, Canada M5S 1A4

wortman@csri.toronto.edu

Abstract

In this paper we describe a collection of techniques for the design and implementation of concurrent compilers. We begin by describing a technique for dividing a source program into many *streams* so that each stream can be compiled concurrently. We discuss several compiler design issues unique to concurrent compilers including source program partitioning, symbol table management, compiler task scheduling and information flow constraints. The application of our techniques is illustrated by a complete design for a concurrent Modula-2+ compiler. After describing the structure of this compiler we present an experimental evaluation of the compiler's performance that demonstrates that significant improvements in compilation time can be achieved through the use of concurrency.

1 Introduction

The Concurrent Compiler Development (CCD) Project at the University of Toronto was a long term effort to explore issues in the design and implementation of concurrent compilers for modern programming languages. We developed techniques for structuring compilers to achieve significant compilation time speedups through concurrent processing. Although we describe a particular prototype concurrent compiler in this paper, the techniques we have developed are applicable to compilers for a broad range of languages.

To provide a realistic environment for developing and evaluating our ideas we built a prototype concurrent compiler [JW90] for the Modula-2+ superset of Modula-2 [Rov86]. Modula-2+ was chosen as the target language partly for reasons of expediency (i.e. availability of an existing compiler that served as a starting point) , but primarily because Modula-2+ is a large modern programming language that presented many challenges to the language implementor. We felt that if we could produce an efficient compiler for Modula-2+ then it would be easy to handle languages that are easier to compile concurrently such as Pascal and C. During the CCD project we implemented seven major variations of our concurrent compiler design to test various implementation strategies. The compiler described in this paper is the "best of 7" compiler that represents the culmination of our efforts.

Our paper in the 1988 PLDI conference [SWJ$^+$88] discussed the narrow issues of semantic analysis and symbol table management in concurrent compilers. A longer discussion of this topic has been published elsewhere [SW91]. This paper describes the design and evaluation of a complete concurrent compiler for the Modula-2+ superset of Modula-2. A more detailed discussion of our prototype compilers has been published as a technical report [JW90].

The research described in this paper differs from previous work in the area in a number of ways:

- It describes a complete, implemented, concurrent com-

*Current address: IBM Canada Ltd., Toronto Laboratory, 844 Don Mills Rd., North York, Ontario, Canada M3C 1V7 piler. In their comprehensive bibliography on Concurrent Compilation [SB90] the authors identified only three implemented modern compilers for shared memory multiprocessors [Van87],[Sch79] and ours. The remaining work that they cite are theoretical studies, paper designs or concern only one phase of compilation (e.g. parallel parsing).

- It goes much further than previous efforts in restructuring compilers to take advantage of concurrent hardware. We have taken a very aggressive approach to partitioning compilation into separately executable tasks and to splitting the source program into separately compilable sub-units. For example, Vandevoorde [Van87] split his compiler into a scanning phase and an "everything else" phase. He did not address the Doesn't Know Yet problem discussed in Section 2.2.

- The symbol table search mechanism (Skeptical Handling) presented in Section 2.2 is a new approach to dealing with the problem of incomplete symbol tables that arises in concurrent compilers. It supersedes our previous work [JW90, SW91].

- The discussion of information flows that constrain concurrent processing in Section 2.4 is a new approach to analyzing the performance of concurrent compilers.

- By careful design and optimization of the entire compiler, we have been able to achieve much better parallel speedups than previous efforts. We have also been able to demonstrate that compilers built using our techniques perform better as the size of the program being compiled increases. The closest comparable effort is Vandevoorde's parallel C compiler[Van87]. On a 5 processor Firefly he was able to achieve a speedup of 2.5 .. 3.3 on large programs and 1.0 .. 2.0 on smaller ones. In the limiting case of a "best" source program we have been able to demonstrate almost linear parallel speedup (see Figure. 4).

Early in the CCD project we decided to concentrate our efforts on compilers for modern block-structured procedural programming languages. Considering a broader class of languages would have diluted our resources. We concentrated on the design of compilers for shared memory multiprocessor systems with modest numbers (i.e. tens not hundreds) of processors. We restricted ourselves to languages in which reserved words were used to determine the lexical structure of programs. This restriction allows us to partition programs for concurrent processing during lexical analysis. Languages which use keywords rather than reserved words to determine program structure usually can't be partitioned for concurrent processing until after syntax analysis thereby reducing the potential for achieving faster compilation through concurrency.

The concurrent compilers that we have designed operate by splitting the source program into separately compilable *streams* that correspond to scopes of declaration in the program. Each stream is then processed through conventional parsing, semantic analysis and code generation phases. At the end of compilation the code generated for all of the streams is merged together to produce the compiler output. The structure of the compiler is discussed in more detail in Section 3. We begin with a discussion of the major design issues that lead to this structure.

2 Compiler Design Issues

During the CCD project we identified four design issues that had a dominant effect on the performance of our concurrent compilers:

1. *Early* splitting of the source program into separately compilable streams and *late* merging of separately compiled object code into a a complete object module.

2. Management of the compiler symbol table and the Doesn't Know Yet Problem.

3. Management and scheduling of the compiler tasks for each stream.

4. Inter-scope flows of information that constrain concurrent processing.

There are many other small design details that must be handled well to achieve maximum compiler performance, but doing well on these four issues is essential.

2.1 Source Splitting and Merging

Opportunities for concurrent processing are increased by splitting the program being compiled into independently compilable streams as early in the compilation as possible. In compiling a Modula-2+ module, separately compilable streams arise from three sources:

- The body of the module being compiled.
- Procedures contained in the module body.
- Interfaces (definition modules) that are directly or indirectly imported by the module.

The requirement that reserved words determine source program structure means that these streams can be identified by a simple finite state recognizer that processes the sequence of tokens produced during or just after lexical analysis. A small amount of token stream lookahead may be necessary to resolve lexical tokens that have multiple interpretations (e.g., PROCEDURE in Modula-2). In our concurrent compiler design the procedure is the smallest program unit that is compiled concurrently. It is a straightforward exercise to generate code for each procedure separately and to merge this code using simple concatenation. Most previous efforts to build concurrent compilers performed splitting during parsing and thereby limited opportunities for parallel processing of the source program. Many paper designs for concurrent compilers proposed splitting the source program being compiled at arbitrary points without adequate consideration for the difficulties that arbitrary splitting would cause for symbol table management and code generation.

2.2 Symbol Table Management and the Doesn't Know Yet Problem

Management of the compiler's symbol table is a critical issue in achieving good performance and correct semantic analysis in a concurrent compiler. The compilation of parts of a program concurrently introduces new problems in symbol table management that do not exist in conventional sequential compilers [SWJ$^+$88, SW91].

In our compiler the units of compilation correspond directly to major scopes of declaration. The main reason for this design decision was to allow the compiler symbol table to be organized on a scope basis [SGR79]. This has the desirable effect of minimizing inter-stream dependencies and making the detection of incomplete tables easier. We use a separate symbol table for each scope of declaration (definition module, main module, procedure). These symbol tables are linked together to provided the correct scope ancestry path for resolving names.

A problem unique to our style of concurrent compilers is the Doesn't Know Yet (DKY) problem [SWJ$^+$88, Ses88, SW91]. In a sequential compiler, a search of the symbol table for a given identifier either succeeds indicating that the identifier is known or fails, indicating unequivocally that the identifier is not known to the compiler (assuming a suitable scheme for handling allowable forward references). In a concurrent compiler there is a third possible result from a search of the symbol table. The identifier may not be found because the symbol table being searched is incomplete[1]. The table that was being searched may be being constructed concurrently by some other compiler task. A concurrent compiler must recognize when this Doesn't Know Yet condition arises and deal with it in a way that doesn't violate the semantics of the language being compiled. Symbol table search must correctly locate declared symbols and never fail to detect an undeclared symbol.

We experimented with a number of strategies for dealing with the DKY problem [Ses88, JW90]. The differences between these strategies are in the amount of delay incurred when a DKY occurs, the amount of concurrent processing that can be achieved, and the effort required to implement the strategy. The list of strategies presented below is roughly ordered by decreasing DKY delay, increasing concurrent processing potential and increasing implementation effort. The major DKY strategies that we investigated are:

- *Avoidance*: delay the start of semantic analysis for a scope until the declaration analysis of its parent scope is complete. This order guarantees that symbol table searches originating in the scope will never encounter an incomplete symbol table in an outer scope.
- *Pessimistic Handling*: symbol table search blocks and

[1]We assume that creation of symbol table entries is atomic with respect to symbol table search so that the possibility of symbol table search finding an incomplete symbol table entry does not arise.

waits for table completion when it encounters an incomplete symbol table in some outer scope.

- *Skeptical Handling*: symbol table search blocks and waits for table completion when it fails to find the identifier it is searching for in an incomplete symbol table in some outer scope.
- *Optimistic Handling*: symbol table search blocks and waits for completion of a symbol table *entry* or the completion of a symbol table (whichever comes first) when it fails to find the identifier it is searching for in an incomplete symbol table in some outer scope.

When a symbol table search blocks, the embedded scheduler in our compiler (See Section 2.3) is notified. It attempts to find another task to run. It will preferentially try to run the task which will resolve the DKY blockage. In our performance testing, the choice of a method for dealing with the DKY problem caused a variation of about 10% in overall compiler performance. The Skeptical Handling algorithm described in Figure 6 is our recommendation for the best compromise between compiler performance and ease of implementation. This algorithm incurs the cost of a duplicate search for identifiers that are found after a DKY blockage, but it gains performance by searching incomplete tables before incurring a DKY blockage. The performance of the Skeptical Handling algorithm is analyzed in Section 4.3.

Another symbol table issue is the handling of builtin names, i.e., identifiers that are automatically provided by the compiler. These names are typically builtin input/output routines or mathematical routines like *sin* and *sqrt*. In a conventional compiler the typical way to implement builtin names is to create a global scope that is logically the parent scope of the module being compiled. The symbol table for this global scope is preinitialized by the compiler to contain all builtin names. With this organization, the normal scope-chaining symbol table search will automatically find the builtin names if they have not been redeclared in the program. This mechanism is not suitable for a concurrent compiler because it causes the first reference to a builtin name to potentially incur DKY waits on all scopes out to the global scope. In particular, the scope corresponding to the module body can be quite large so an unnecessary DKY wait on this scope can cause a significant delay. We took advantage of the fact that builtin names could not be redefined in Modula-2+ and treated builtin names as if they were declared local to each scope. This approach did not require any replication of symbol table entries, instead it was done by a simple modification of the symbol table search mechanism.

2.3 Compiler Task Scheduling

In our compiler structure, we exploit parallelism on two levels. First, as already mentioned, the compilation is split into streams which are processed in parallel. There is one stream for the main module, one for each procedure stream defined therein and one stream for each definition module imported directly or indirectly by the module. Second, we exploit parallelism within the processing of a stream by partitioning the stream into a number of *tasks*, which correspond to the traditional phases of compilation. The number and type of the tasks a particular stream is partitioned into depends not only on the type of the stream but also on the version of the compiler. There are between 2 and 5 tasks per stream (see Figure 5). The task types of the skeptical handling compiler are described in section 3. The task types were chosen to partition the compilation as quickly as possible, in an effort to get all the compiler's worker threads busy. The early tasks are large in number but, in general, small, while the later ones, such as the statement analyzer/code generator tasks, are large but fewer in number. However, there is no need to partition these later tasks further (although there may be parallelism that can be exploited) because there are almost always enough of these tasks to ensure that all processors are fully utilized.

2.3.1 Tasks

The task is the atomic unit of parallelism in our compilers. The execution of a given task is constrained by the rate the tasks on which it depends for information are progressing (See sections 2.2 and 2.4). These constraints are embodied by *events*, the concurrency mechanism used. An event is simply something that either has or has not occurred. A task waits on an event if and only if it hasn't occurred. Events are discussed further in section 2.3.3 below.

A task communicates with the tasks on which it depends, and which depend on it, as a producer/consumer pair on a particular data structure. Each such data structure has associated with it one or more events, signaled by the producer when the events occur. The signaling of such an event indicates that a particular portion of a shared data structure is complete and ready to be used. As an example, the Splitter task and the Lexor task of a main module stream communicate via a lexical token queue. The elements in this queue are blocks of tokens. Each block is associated with one event. When the Lexor fills a token block, the block's event is signaled, indicating to the Splitter that it now may begin to read the tokens of that block.

2.3.2 The Supervisors Approach

The number of tasks in a compilation ranges from a dozen or so in a very small compilation to a few hundred in a large compilation. We concur with the results of Vandevoorde and Roberts [VR88] that the best performance on the type of multiprocessors we are considering results from carefully limiting the amount of concurrent activity in the system so as not to overwhelm machine resources. If an attempt was made to execute all of these tasks concurrently saturation of the operating system and the virtual memory would make concurrent compilation effective only for small source modules. Instead, we initiate one compiler process (Worker) for each real hardware processor. These workers are managed by a *supervisor* which oversees the assignment of tasks to workers. This approach, called "Supervisors", is an extension of WorkCrews [VR88] that handles blockable tasks.

At compiler initialization time, the tasks of the main module stream are created by the compiler initialization thread, and then a number of worker threads corresponding to the number of hardware processors are created. The initialization thread then blocks, waiting for the workers to perform the compilation. The workers begin by searching the supervisor's task queuing structure for tasks to perform. One begins the main module stream's Lexor task, and the compilation begins.

2.3.3 Events

Events are classified into three categories. First, *avoided* events are those events that must occur before the task(s) that will wait on them can begin. They are called this because we avoid waiting on the event by simply not assigning the task to a worker until the event has occurred. The rationale is that the amount of work a task could do before being forced to wait on such an event is so little so as to be outweighed by the scheduling cost when the wait occurs.

The second type of event is the *handled* event. We allow tasks that may wait on handled events to begin, hoping that the amount of work they can do before being forced to wait on such an event outweighs the scheduling cost should the wait occur. If the wait does occur, the worker performing the task searches for other tasks to execute, showing preference to those tasks whose execution will lead toward the event occurring.

Lastly, *barrier* events are a special kind of handled event for which the worker executing the waiting task is not rescheduled. The worker simply waits for the event to occur. Barrier events are only used in the token streams. We know it is safe to simply wait because token consumers (the Splitter task and the parser tasks of definition module streams) are only started once their corresponding Lexor tasks have begun, and Lexor tasks never block. Thus, the possibility of deadlock is avoided. Our measurements show that the delays due to workers waiting on barrier events are quite small in typical compilations.

The tasks and events of the different major versions of our compilers vary primarily in how DKY events are treated. In the Avoidance compiler, there is one DKY event per symbol table, and it is an avoided event. This means that the tasks of a stream that depend on another stream's symbol table are not begun until that symbol table is complete. In the Pessimistic Handling compiler, there is again one DKY event per symbol table, but each is a handled event. Here, tasks are allowed to begin regardless of the completion status of the symbol tables on which they (may) depend, but must block as soon as they try to look up a symbol in an incomplete symbol table. These events are signaled when their corresponding symbol table is completed. A refinement of this strategy is presented in a Skeptical Handling compiler, where a searching task is allowed to look into an incomplete symbol table and will block only if the symbol is not found. As before, the blocked task is signaled when the symbol table is completed, and it attempts again to find the symbol in the now complete table.

The Optimistic Handling compiler implements a different scheme. Here, there is one DKY event per symbol. In this method, when a task looks up a symbol, it waits on the symbol's DKY event (should it be in the process of being defined), or creates an empty symbol with the name being searched for (if it isn't in the table) and waits on that symbol's event. If the symbol is really in that scope, the task is signaled and allowed to proceed. Otherwise, when the table is completed, it is traversed and all unsignaled events (corresponding to symbols that are being looked up in the table but are not defined in the scope) are signaled, allowing blocked tasks to continue searching. While this method allows the maximum parallelism to be exploited, and indeed has the best self-relative speedup, the overhead of maintaining so many events outweighs the advantages of the technique.

2.3.4 Worker Scheduling

The role of the Supervisor is to assign tasks to workers in such a manner as to minimize compilation time. This is accomplished in two ways. First, free workers are assigned tasks in an intelligent manner, attempting to select the tasks whose execution will benefit the most tasks. Second, handled events are implemented in such a way that allows workers whose tasks are blocked on handled events perform other tasks.

Task assignment is optimized in two ways. First, avoided events prevent a task that will have to wait almost immediately from being selected for execution. Also, the Supervisor implements the list of tasks waiting to be executed as a priority queue. In the Skeptical Handling compiler, the queues are searched for new work in the following way:

1. Lexor tasks
2. Splitter task
3. Importer tasks
4. Definition Module Parser/Declarations Analyzer tasks
5. Module Parser/Declarations Analyzer task
6. Procedure Parser/Declarations Analyzer tasks
7. Long Procedure Statement Analyzer/Code Generator tasks
8. Short Procedure Statement Analyzer/Code Generator tasks

Code is generated for long procedures before short ones to avoid a long sequential tail at the end of the compilation, as one worker struggles to generate code for one long procedure after finishing a number of short ones and all the other workers are finished.

When a task waits on a handled event, its associated worker does not simply block but instead is assigned another task that it is eligible to execute. This task is always the one that signals the event on which the worker is waiting, unless that task has been started by another worker.

This leads to an interesting problem. A task that is begun by a particular worker must be completed by that worker, even if it blocks during execution. This is required by the underlying implementation of threads. Thus, the set of tasks a blocked worker is eligible to execute consists of only those tasks that do not block on any event not yet signaled by some task in the set of tasks being executed the worker (who are all currently blocked), otherwise deadlock may occur.

2.4 Inter-scope Information Flows

The implementation of block structured programming languages requires that information in the compilers symbol table be shared among different scopes. By processing scopes concurrently we introduce constraints on this sharing that do not exist in a sequential compiler. Delays waiting for this shared information can constitute a major impediment to concurrent processing.

An example of an information flow constraint arises in the processing of procedure headings in Modula-2+. A procedure heading contains the procedure's name, the name and specification for each of the procedure's formal parameters. The information in the procedure header needs to be shared between the parent scope in which the procedure is declared where it is used to validate and generate code for calls on the procedure and the child scope (the body of the procedure) where it is used to validate and generate code for access to the procedure parameters. In sequential compilers the sharing of this information occurs automatically given the usual order of program processing and symbol table construction. The information that is shared consists of the compiler symbol table entry for the procedure and the entries for its parameters.

We investigated three alternatives for implementing this particular case of sharing in our concurrent compiler:

1. Process the procedure heading in the parent scope and copy the symbol table entries generated by this processing into the symbol table for the child scope.

2. Process the procedure heading in the child scope and copy the symbol table entries produced into the parent scope.

3. Process the parameter heading separately in the parent and child scopes taking care to guarantee that identical symbol table entries are produced in both scopes.

In the first alternative, processing of the child scope should be delayed until the procedure header has been processed in the parent scope. This synchronization is in effect a DKY blockage from the child scope to its parameter definitions in the parent scope. Since this DKY is inevitable and since very little work can be done on a procedure stream until the procedure's parameters have been processed, we found it more efficient to change this particular instance of a handled DKY event into an avoided event by delaying processing the child scope until the parent scope had completely processed the procedure heading[2]. The second alternative is very difficult to implement without introducing the possibility of a parent/child deadlock. The third alternative eliminates the information sharing between parent and child scopes. It is the cleanest solution, however in our evaluation it was about 3% slower than the first alternative due to redundant effort.

[2]Procedures with no parameters could be started earlier as a special case, but we found them to be so infrequent that the extra effort was not justified

3 Compiler Structure

The unit of compilation is a Modula-2+ module (M) that is physically represented as a definition module file (M.def) and an implementation module file (M.mod)[3]. The file M.def contains declarations for the interface between M and its clients. The file M.mod contains declarations for constants, variables, types and procedures that are the implementation of M.

The task structure of our concurrent compiler is shown in Figure 5 The three columns in the figure show the compiler tasks that are applied to the streams for definition modules, implementation modules[4] and procedures respectively. The tasks in the left column are applied to each definition module that is imported directly or indirectly by the module. The tasks in the right column are applied to each procedure in the module. We have used an unorthodox task division in the middle and back part of the compiler as a part of our strategy for dealing with DKYs. One compiler task performs syntax analysis on the entire stream and semantic analysis on declarations as would be done in a traditional sequential compiler. A parse tree is built for statements, but semantic analysis of statements is deferred to a subsequent task that performs semantic analysis on statements and then generates code for the statements. The symbol table for the declarations is marked complete before the statment parse tree is built. Building the statement parse tree is sufficiently fast that the added complexity of parsing statments separately was not considered. The rationale behind this division is that fast processing of the declaration parts of streams will assist in resolving DKY blockages by causing symbol tables to be completed earlier in the compilation. By the time the statement analysis and code generation tasks are ready to run there is usually many more parallel tasks available to run than there are processors on which to run them, so we incur no loss in processing efficiency by combining statement semantic analysis with code generation in a single task.

The compilation of module M begins with the lexical analysis of the file M.mod. The compiler optimistically anticipates the existence of a file M.def and tries to start

[3]We ignore the relatively rare self contained program module

[4]the import task for the implementation module is shown in the left column for convenience.

processing this file as soon as possible. The token stream produced by the lexical analysis phase is directed to the splitter and import tasks. The import task searches the token stream for IMPORT declarations and starts a new stream for each imported definition module that it discovers. It starts a lexical analysis, importer, syntax and semantic analysis sequence for each new definition module stream. The token stream produced by the lexical analysis of each imported definition module is also fed into its import task to detect indirectly imported interfaces. A "once-only" table is used to guarantee that each definition module referenced in a compilation is processed exactly once.

The splitter task searches for the reserved word PROCEDURE in the token stream of M.mod. It creates a new stream for each procedure it detects and diverts the lexical tokens for the procedure to that stream. It then starts a syntax analysis, semantic analysis, code generation sequence for that stream. The main module body which has now been striped of all embedded streams is processed through syntax analysis, semantic analysis and code generation. At the end of compilation, a merge task concatenates the output of separate code generation streams to form the complete compiler result. Because the unit of merging is the code for an entire procedure, this concatenation can be done in any order and concurrently with other compiler activity.

4 Compiler Performance

We have conducted numerous experiments to evaluate the performance of our prototype concurrent compilers. In this section we summarize the results of those experiments.

4.1 Test Environment

The concurrent compiler was written in Modula-2+. The compiler runs on DEC Firefly prototype workstations [TSS88] under the Topaz operating system [MS87]. The Firefly used to obtain the results presented here was configured with 64 Mb of main memory and eight CVax processors. The compiler performance results presented in this section should be view in the context of the known performance limitations of the Firefly hardware. At high levels of concurrent activity, memory bus saturation effects and fixed processor priorities for access to memory degrade the performance of all processors [TSS88]. Topaz provides a lightweight threads mechanism within a single address space. This mechanism was used extensively within the compiler.

The suite of 37 programs used to evaluate our compiler were taken from a very large library of Modula-2+ software that was made available to us by the DEC Systems Research Center. The programs in the test suite represent the efforts of a large number of authors and include a variety of programming styles. The programs in the test suite were chosen at the start of the CCD project to be a representative sample of the load that might be seen by a typical compiler. The programs in the test suite range from very small to very large. The gross characteristics of the test suite are given in Table 1. In this table *Imported Interfaces* are the number if definition modules imported directly or indirectly by the module being compiled. *Import Nesting Depth* is the maximum length of the import nesting chain for each module. More details on the test suite are available in [SW91, JW90].

Attribute	Minimum	Median	Maximum
Module size (bytes)	2,371	13,180	336,312
Seq. Compile Time (sec)	2.30	10.27	107.85
Imported Interfaces	4	17	133
Import Nesting Depth	1	5	12
Number of Procedures	2	16	221
Number of Streams	15	37	315

Table 1: Description of Test Suite

4.2 Speedup Results

To evaluate the improvement in compilation time that we were able to achieve through concurrent processing, we evaluated our concurrent compiler against a traditional sequential compiler for Modula-2+ and against itself. All test runs were made on an otherwise idle Firefly workstation. The results presented are the average of 10 runs. The run-to-run variation in our timing results was very small, on the order of 1%. Running on one processor, the concurrent compiler was 4.3% slower than the sequential compiler. This is due to the extra processing that was introduced to achieve concurrency which is wasted on a single processor. Figure 1 shows the self-relative speedup of our compiler as it was run on 1 through 8 processors[5]. To test the limits of our technique, we mechanically generated a synthetic module (Synth.mod) that would have the best possible speedup for our technique. This module has been constructed so that it generates ample parallel work for the compiler and never incurs a DKY blockage. The speedup results for the synthetic module are shown in Figure 2. For comparison we have included a reference line for linear speedup and results for the human-authored module that had the best overall speedup in the test suite.

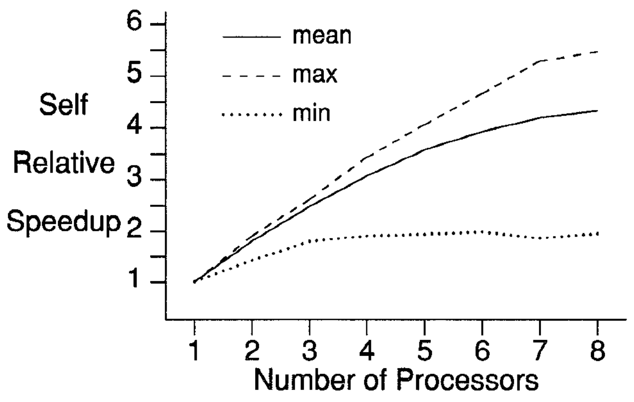

Figure 1: Test Suite Self Relative Speedup

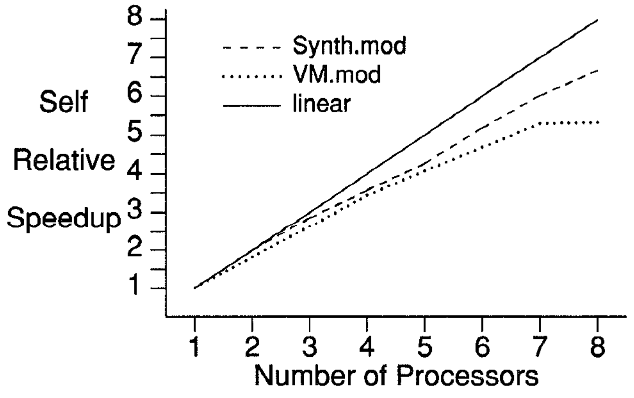

Figure 2: Best Case Self Relative Speedup

To get an indication of how our techniques scale as a function of the size of module being compiled, we divided the programs in the test suite into approximate quartiles based on their 1-processor compilation times: 0..5 seconds (10 programs), 5..10 seconds (8 programs), 10..30 seconds (10 programs) and 30..109 seconds (9 programs). Figure 3 shows the self-relative speedup data plotted separately for the programs in each quartile. From these plots we can see that the speedup obtainable through concurrent processing is limited for small programs, but increases as the size of program being compiled increases. For programs with a sequential compile time of less than 10 seconds, a speedup of 2.5 is obtained with 4 processors and there is no real advantage in using more processors. It is evident from the first two compilations in Figure 6 that small modules don't generate enough parallel work to fully utilize the multiprocessor. For larger programs (sequential compile time greater than 10 seconds), the speedup continues to increase although at a decreasing rate as more processors are used. We attribute this to the greater amount of parallel work available in these programs. As a graphical illustration of our compilers operation, we present the WatchTool snapshots in Figure 4. This figure shows processor activity (vertical axis) as a function of time (horizontal axis) as a program from each of the quartiles was compiled on an eight processor Firefly. There is a ten second delay between compilations. For comparison purposes, the rightmost peak in this figure is a compilation of the synthetic module that was described above.

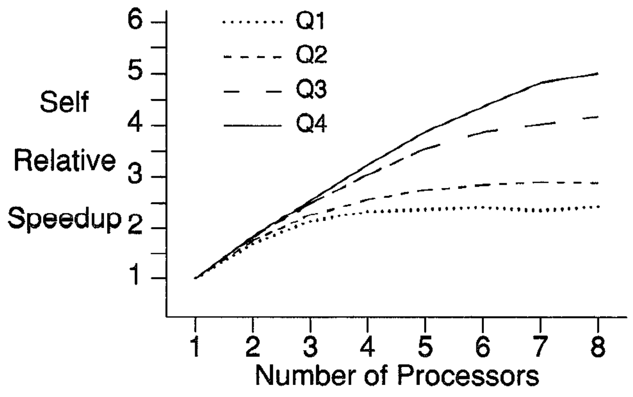

Figure 3: Speedup by Quartiles

4.3 Analysis of Skeptical Handling

The skeptical handling symbol table lookup mechanism described in Section 2.2 is an attempt to gain performance by

[5]Table 3 contains the data used to generate Figures 3,4 and 5

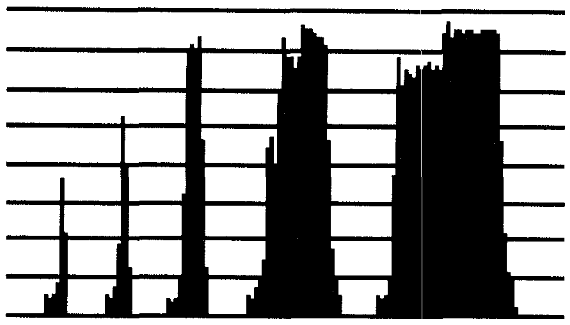

Figure 4: WatchTool Snapshot

searching incomplete symbol tables. To measure the effectiveness of this strategy, we gathered performance data for this mechanism for one compilation of our test suite. The lookup mechanism is applied in two circumstances: search for a simple identifier occurring in a procedure or the main module, and search for a qualified identifier (i.e. an identifier explicitly associated with an imported definition module). The statistics for identifier search are given in Table 2. The "Found when" column indicates the conditions under which the identifier was found. "Search" indicates that the identifier was found during an outward search through the scope parentage chain. "After DKY" indicates that the identifier was found in a scope that was completed after a DKY blockage. The "scope" column indicates the scope in which the identifier was found. "Self" is the scope of the stream that initiated the search, "other" is some other explicitly designated initial search scope, "outer" is a scope that was accessed chaining outward through the scope parentage chain, "WITH" is the scope of a WITH statement and "Builtin" is the scope containing all identifiers predefined by the compiler. The "completeness column" indicates that state of the scope at the start of the search. The last two columns given the number and percentage of lookups that succeeded for the particular combination of Found, scope and completeness. The interesting cases in this table are the searches that found an identifier in an incomplete table. This is the case where Skeptical Handling has an edge over more conservative strategies. The higher frequency of search success in incomplete tables and DKY blockages in the case of qualified names can be attributed to to the higher usage of qualified names in declarations. The majority of simple identifier references arise from the statements in the bodies of procedures. By the time these are processed there is a high probability that the symbol tables for outer scopes and definition modules have been completed. This table also shows that blockage due to the DKY condition is relatively rare. The low frequency of DKY blockages is one reason why the choice of a DKY strategy has only a small effect on overall compiler performance.

4.4 An Activity View of Concurrent Compilation

In the early part of a compilation the dominant activity is the lexical, syntax and semantic analysis of definition modules. The definition modules imported directly and indirectly by the main module form a tree. The need to resolve DKY blockages quickly and the task scheduling strategy used by our scheduler typically causes this tree to be processed in a bottom up order. At this point, the streams corresponding to the procedures in the main module have been identified by the splitter but processing of these streams is deferred until the corresponding procedure headings have been processed in the main module stream. Once processing of the main module stream reaches the procedure headers, a large amount of parallel work becomes available as processing of many procedure streams can be initiated. From this point on a high level of concurrent processing is possible through to the end of the compilation.

Figure 7 illustrates the compiler activities in a typical concurrent compilation. The horizontal axis is time, the vertical axis is processor number for an eight processor Firefly. The bars in this figure indicate different kinds of compiler activity. The dark gray bars at the left side are lexical analysis tasks. The light gray bars in the middle are parser/declaration semantic analysis tasks. The darker gray bars on the right side of the figure represent statement analysis/code generation tasks. The light gray bar for processor 7 is the splitter task. At the scale of this figure, the importer and merge tasks are too small to be easily visible. The activity lull in the center of this figure is caused by delays waiting for DKYs to be resolved and by the delay involved in waiting for procedure headers to be processed in the main module body as described in Section 2.4.

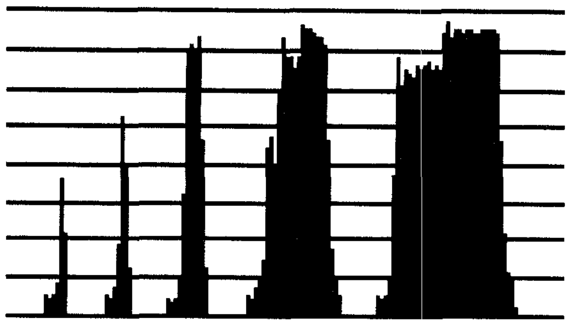

Figure 4: WatchTool Snapshot

searching incomplete symbol tables. To measure the effectiveness of this strategy, we gathered performance data for this mechanism for one compilation of our test suite. The lookup mechanism is applied in two circumstances: search for a simple identifier occurring in a procedure or the main module, and search for a qualified identifier (i.e. an identifier explicitly associated with an imported definition module). The statistics for identifier search are given in Table 2. The "Found when" column indicates the conditions under which the identifier was found. "Search" indicates that the identifier was found during an outward search through the scope parentage chain. "After DKY" indicates that the identifier was found in a scope that was completed after a DKY blockage. The "scope" column indicates the scope in which the identifier was found. "Self" is the scope of the stream that initiated the search, "other" is some other explicitly designated initial search scope, "outer" is a scope that was accessed chaining outward through the scope parentage chain, "WITH" is the scope of a WITH statement and "Builtin" is the scope containing all identifiers predefined by the compiler. The "completeness column" indicates that state of the scope at the start of the search. The last two columns given the number and percentage of lookups that succeeded for the particular combination of Found, scope and completeness. The interesting cases in this table are the searches that found an identifier in an incomplete table. This is the case where Skeptical Handling has an edge over more conservative strategies. The higher frequency of search success in incomplete tables and DKY blockages in the case of qualified names can be attributed to to the higher usage of qualified names in declarations. The majority of simple identifier references arise from the statements in the bodies of procedures. By the time these are processed there is a high probability that the symbol tables for outer scopes and definition modules have been completed. This table also shows that blockage due to the DKY condition is relatively rare. The low frequency of DKY blockages is one reason why the choice of a DKY strategy has only a small effect on overall compiler performance.

4.4 An Activity View of Concurrent Compilation

In the early part of a compilation the dominant activity is the lexical, syntax and semantic analysis of definition modules. The definition modules imported directly and indirectly by the main module form a tree. The need to resolve DKY blockages quickly and the task scheduling strategy used by our scheduler typically causes this tree to be processed in a bottom up order. At this point, the streams corresponding to the procedures in the main module have been identified by the splitter but processing of these streams is deferred until the corresponding procedure headings have been processed in the main module stream. Once processing of the main module stream reaches the procedure headers, a large amount of parallel work becomes available as processing of many procedure streams can be initiated. From this point on a high level of concurrent processing is possible through to the end of the compilation.

Figure 7 illustrates the compiler activities in a typical concurrent compilation. The horizontal axis is time, the vertical axis is processor number for an eight processor Firefly. The bars in this figure indicate different kinds of compiler activity. The dark gray bars at the left side are lexical analysis tasks. The light gray bars in the middle are parser/declaration semantic analysis tasks. The darker gray bars on the right side of the figure represent statement analysis/code generation tasks. The light gray bar for processor 7 is the splitter task. At the scale of this figure, the importer and merge tasks are too small to be easily visible. The activity lull in the center of this figure is caused by delays waiting for DKYs to be resolved and by the delay involved in waiting for procedure headers to be processed in the main module body as described in Section 2.4.

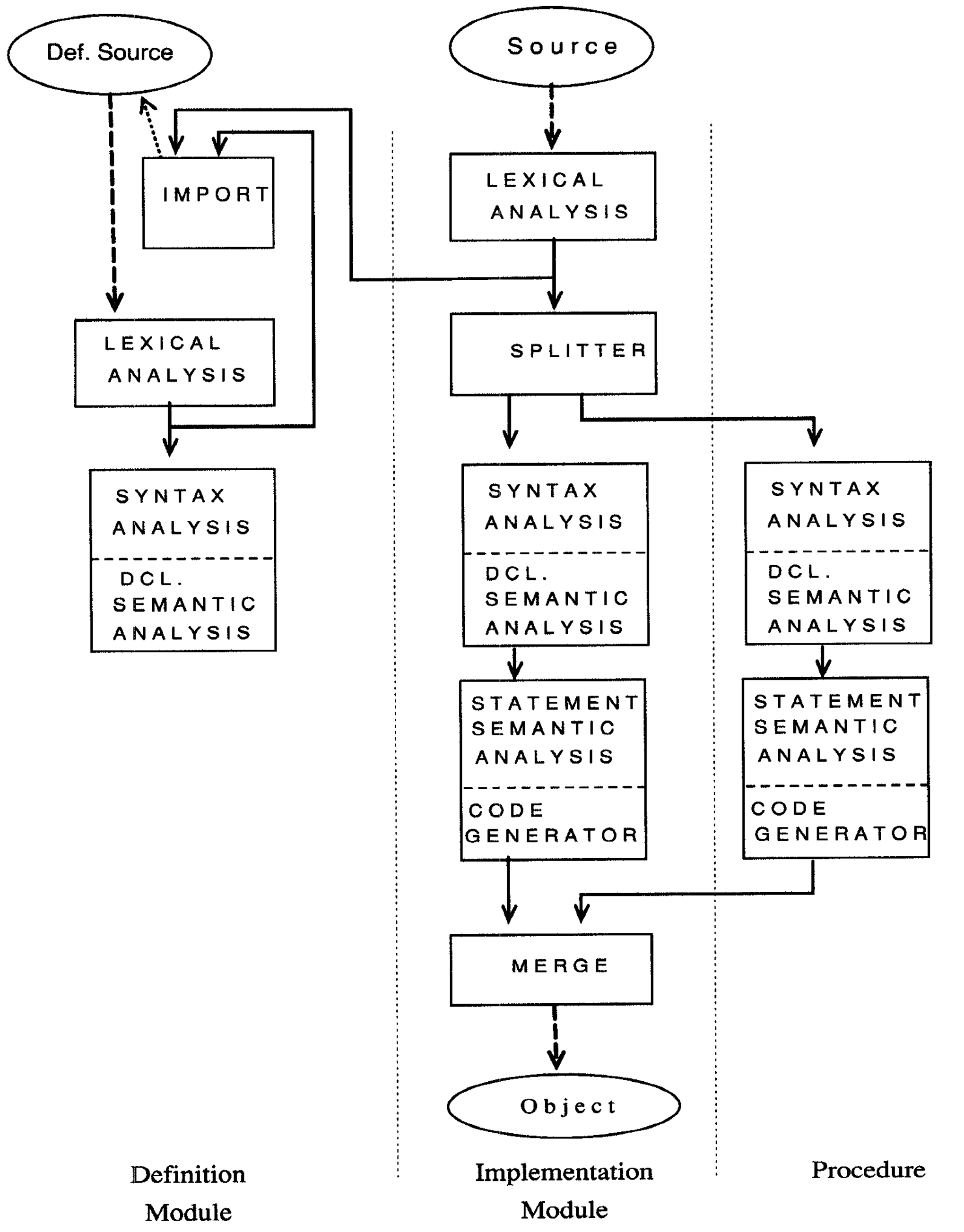

Figure 5: Compiler Task Structure

Simple Identifier					Qualified Identifier			
Found when	scope	completeness	number	%	Found when	completeness	number	%
First try	self	complete	52,544	57.87				
First try	other	complete	5,457	6.01				
Search	outer	incomplete	3,228	3.55	First try	incomplete	505	4.00
Search	outer	complete	12,879	14.18	First try	complete	11,775	93.30
After DKY	outer	complete	69	0.08	After DKY	complete	341	2.70
First Try	WITH	complete	2,699	2.97				
First Try	Builtin	complete	13,744	15.14				
Never			184	0.20				

Table 2: Identifier Lookup Statistics

N	Test Suite			BestCase		Quartiles			
	Min	Mean	Max	Synth	VM	Q1	Q2	Q3	Q4
2	1.42	1.81	1.91	1.99	1.81	1.68	1.70	1.81	1.84
3	1.80	2.49	2.62	2.85	2.62	2.13	2.27	2.50	2.55
4	1.91	3.07	3.43	3.57	3.43	2.34	2.57	3.05	3.24
5	1.94	3.58	4.07	4.26	4.07	2.38	2.76	3.56	3.88
6	1.99	3.93	4.67	5.18	4.67	2.42	2.85	3.88	4.37
7	1.86	4.20	5.29	6.01	5.29	2.36	2.91	4.03	4.83
8	1.95	4.34	5.47	6.67	5.32	2.43	2.89	4.19	5.02

Table 3: Summary of Speedup Data

```
Record the completion state of the scope's symbol table
Search the scope's symbol table for the identifier.
   If identifier was found then
        exit with success.
   If the symbol table was initially incomplete then
       Wait for the symbol table to be completed.
       Search the scope's symbol table for the identifier.
       If identifier was found then
           exit with success.
If current scope is outermost scope then
    exit with failure
else
    Continue search in next outermost scope
```

Figure 6: Skeptical Handling Symbol Table Lookup

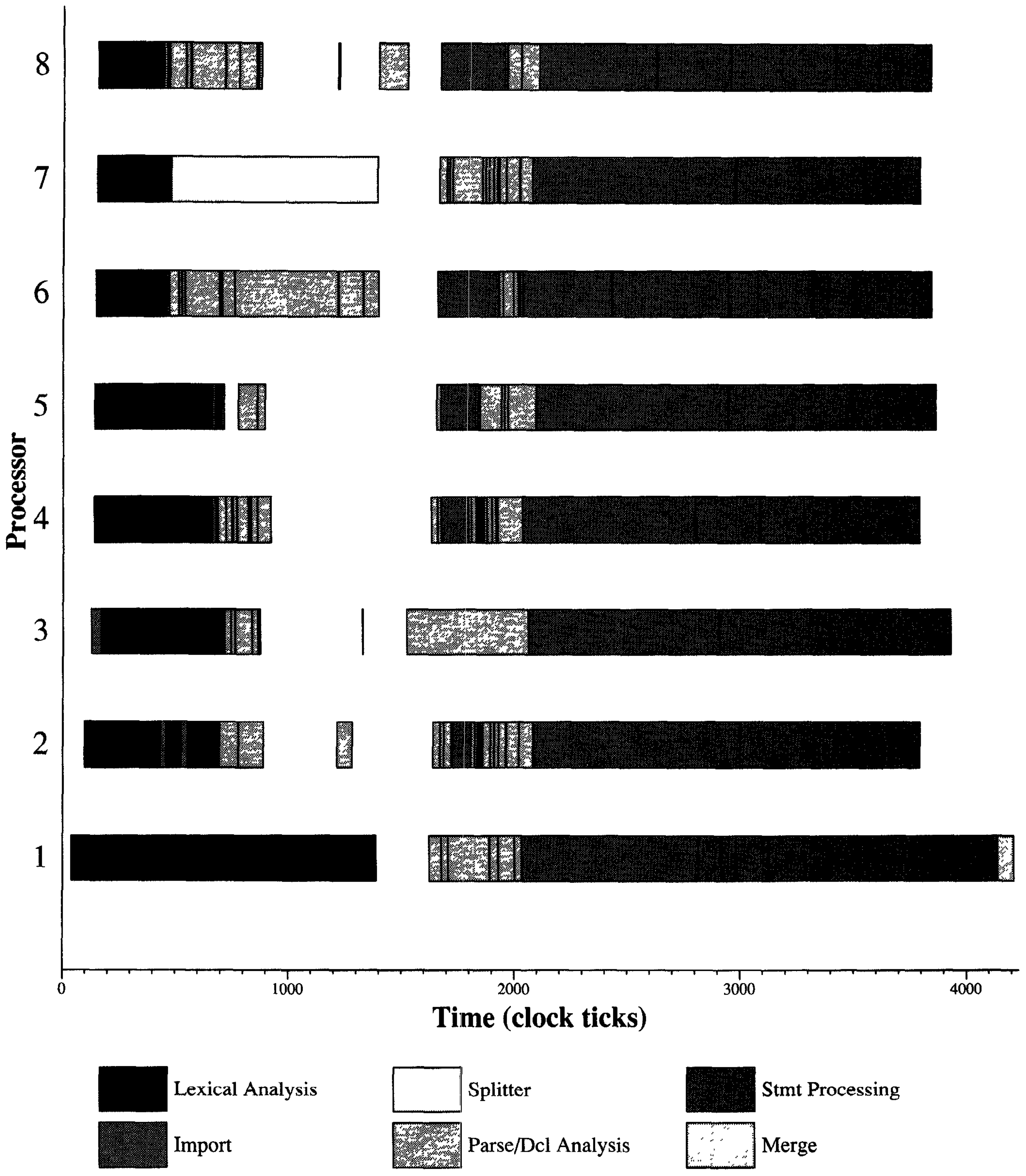

Figure 7: Concurrent Compiler Processor Activity