CODE GENERATION AND SEPARATE COMPILATION IN A PARALLEL PROGRAM DEBUGGER

by

Jong-Deok Choi
Barton P. Miller

Computer Sciences Technical Report #874

August 1989

Code Generation and Separate Compilation in a Parallel Program Debugger

Jong-Deok Choi
choi@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

The *Parallel Program Debugger (PPD)* allows a programmer to find bugs by following dynamic dependences in a program's execution; this technique is called *flowback analysis*. Flowback analysis requires the tracing of all variable references and modifications. PPD avoids the overhead of this tracing by recording only a subset of the program's state during execution, and incrementally filling in the missing details when the programmer makes queries about execution dependences. There is a trade-off between overhead of the tracing during program execution and the speed of generating the missing details during user queries.

Our compiler is divided into four phases. This separation of phases allows us to first compile separate files, and to generate code for these files. Second, we perform interprocedural analysis using the data structures generated by the first phase. Third, we modify the individual assembly files to account for optimizations to the tracing, and to generate tracing for shared variables. The last phase links together all the individual files.

The compiling costs for our debugging system are several times higher than the standard system compiler. Initial results show that there is a large execution time cost savings by carefully selecting the data to be traced during program execution. The savings includes both the size of the trace file and execution time overhead. The most sensitive areas seem to involve small procedures, long-running loops, or arrays.

Research supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

To appear in *Research Monographs in Parallel and Distributed Computing*.

1. Introduction

Separate compilation introduces significant complexity to compilers that perform interprocedural data-flow analysis for optimization [5] and for automatic parallelization [1]. This complexity often defeats the benefits of separate compilation. The *Parallel Program Debugger (PPD)*, a debugging system for parallel programs running on shared memory multi-processors (SMMP) [3, 9], does dependence and data-flow analysis similar to parallelizing compilers, and must also contend with separate compilation.

PPD uses *flowback analysis*[2] to provide information on the causal relationships between events in a program's execution without re-executing the program during debugging. In flowback analysis, the programmer sees, either backward or forward, how information flowed through the program to produce the events of interest. In this way, the programmer can easily locate bugs that led to the detected errors. By using a method called *incremental tracing*[9], PPD is able to keep the program execution overhead of applying flowback analysis relatively low, allowing us to generate only a small amount of trace during program execution. The cornerstone idea of the incremental tracing is to generate a small amount of information, called a *log*, during execution. We then incrementally fill the gap, during the interactive portion of the debugging session, between the information gathered in the log and the information needed to do the flowback analysis using the log.

The log allows us to transfer the cost of generating traces from execution time to debugging time, and also partly to compilation time since we generate static information during compilation. The log also enables us to detect data races in an execution instance of a parallel program and to ensure repeatable execution behavior of a race-free parallel program.

We reduce the run time overhead of producing the log by applying interprocedural analysis [5] and data-flow analysis [7] techniques commonly used in optimizing compilers. However, there is a trade-off between the trace size during execution and response time during debugging [9]; in general, small log size means low execution overhead, but with long response time during debugging. While the size of log should be small enough so as not to introduce an unacceptable performance degradation during execution, it should also be large enough so as not to introduce unacceptable time delays in producing detailed traces during the debugging phase. This paper describes the mechanisms used by PPD to apply these techniques to separately compiled program modules.

In this paper, we are addressing the class of parallel programs that use explicit synchronization primitives, such as the semaphore, monitor, or Ada rendezvous. While we are not addressing automatic parallelism, many of our techniques might be extended to such systems. The techniques in this paper are described in terms of the C programming language [8], but they should generalize to many procedural languages. We address a large part of the C language, including primitives for synchronization, but we do not discuss pointers and dynamic (heap) variables; that is a topic of current investigation.

This paper is organized as follows. Section 2 presents an overview of the strategies used by PPD for debugging parallel programs, and Section 3 describes the run-time traces we generate. Section 4 describes the PPD compiler in detail. We provide performance measurements of the various parts of PPD in Section 5. Section 6 summarizes the paper.

2. Overview of PPD

This section presents an overview of the PPD design. The various components work together to provide an efficient way to debug parallel programs. The goal of the PPD design is to minimize execution time overhead without unduly burdening the other phases of program debugging.

2.1. Flowback Analysis and Incremental Tracing

Flowback analysis would be straightforward if we were to trace every event during the execution of a program. However, doing so is expensive in time and space. The user needs traces for only those events that may have led to the detected error. The problem is that there is no way to know what errors will be detected before the execution of the program; either the user has to generate a trace of every event so that the traces will not lack anything important when an error is detected, or the user has to re-execute a modified program that generates the necessary traces after an error is detected. The first option is expensive, and most often not practical for parallel programs because of unacceptable changes the debugger would introduce in the timing of the interactions between processes. The second option is not practical for programs that lack reproducibility, as is often the case with parallel programs.

We use incremental tracing to overcome the above difficulties. We generate coarse-grained traces, called the log, during program execution. Then, during the interactive portion of the debugging session, we use the log and other compiler-generated information to incrementally produce the fine-grained traces needed to do flowback analysis. This method transfers execution time costs into compile time and debug time. During compile time, we use semantic analyses, such as interprocedural analysis and data flow analysis, to help reduce the amount of information that needs to be generated during program execution. During debug time, we amortize the cost of generating the fine traces over the interactive debugging session. The traces are generated as the programmer asks about dependences in the program.

2.2. Three Phases in Debugging

We divide debugging into three phases: *preparatory* phase, *execution* phase, and *debugging* phase. In the remainder of this section, we briefly describe each of these three phases. A more detailed description of the three phases is given in [4, 9].

Preparatory Phase

During the preparatory phase, the *PPD Compiler* produces, along with the *object code*, the following:

1) the *emulation package* that will generate traces during the debugging phase to fill the gap between the information contained in the log generated during execution phase and the information needed to do flowback analysis;
2) the *static program dependence graph (static graph)* that shows the static (possible) data and control dependences among components of the program; and
3) the *program database* that contains information on the program text such as the places where an identifier is defined or used.

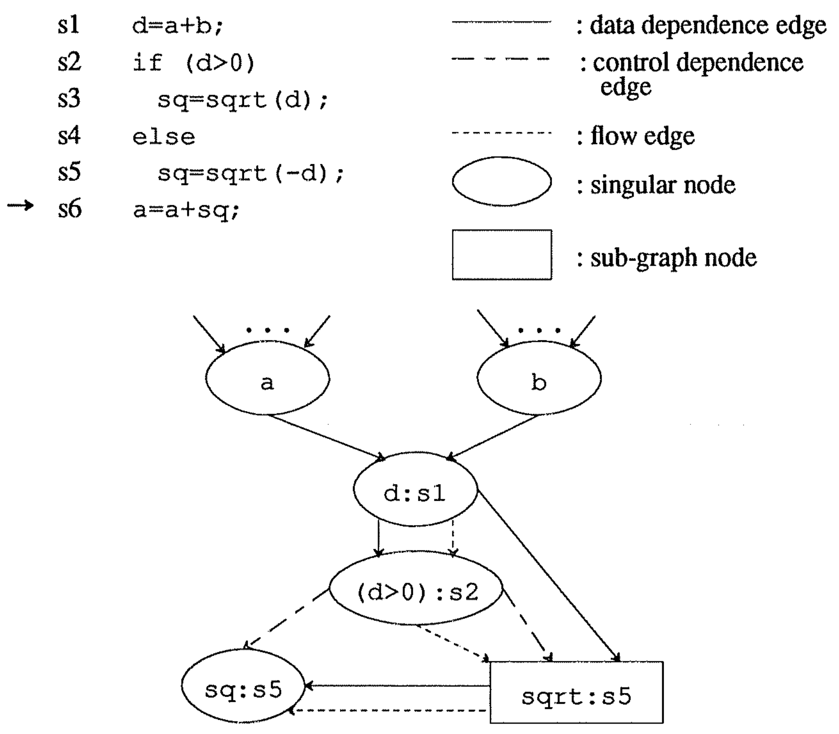
A more detailed description of the static graph is given in [3, 4].

Execution Phase

During this phase, the object code generates the normal program output and a log that contains dynamic information about program execution. The log is used, along with the emulation package, during the debugging phase to generate fine traces for the flowback analysis. The log entries include *prelogs*, which contain the values of the variables that might be read before the next logging point, and *postlogs*, which contain the changes in the program state since the last logging point. The log entries and tracing are described in more detail in Section 3.

Debugging Phase

The goal of the debugging phase is to build a graph of the dynamic dependences in an execution instance of a program, called the *dynamic program dependence graph (dynamic graph)* [9]. Figure 2.1 shows an example dynamic graph. The debugging phase assembles information from the previous phases: the static graph and program database generated during the preparation phase, and the log generated during the execution phase. This information is used by the *PPD Controller*, together with the emulation package, to generate the detailed traces needed to build the dynamic graph.



```
   s1   d=a+b;
   s2   if (d>0)
   s3     sq=sqrt(d);
   s4   else
   s5     sq=sqrt(-d);
→  s6   a=a+sq;
```

Figure 2.1. An Example Dynamic Graph

The Controller also builds the *parallel dynamic program dependence graph* (or *parallel dynamic graph*) [9] to debug parallel programs. The parallel dynamic graph is a subset of the dynamic graph that shows the interactions between processes while hiding the detailed dependences of local events.

3. Run-Time Traces

In this section, we describe the run-time trace (log) in more detail. First, we describe the run-time trace we generate for non-shared variables. Then, we describe the run-time trace we generate for shared variables.

3.1. Non-Shared Variables

We amortize the cost of generating detailed traces, needed for flowback analysis, over the interactive debugging session. The traces are generated incrementally as the programmer asks about the dependences in the program.

As described in Section 2, two types of log entries generated at run time are the prelogs and postlogs. The object code generated by the PPD compiler contains code to generate these log entries. By using semantic analysis, we divide the program into numerous segments of code called *emulation blocks (e-blocks)* [9]. A subroutine is a good example of an emulation block. An e-block is also the unit of incremental tracing during debugging.

The *USED* set of an e-block is the set of variables that might be read-accessed by statements of the e-block. The USED set is computed by a flow-insensitive analysis; a variable in the set might not actually be read-accessed during the execution of the e-block. The *DEFINED* set of an e-block is the set of variables that are write-accessed by statements of the e-block, and is also computed by a flow-insensitive analysis. Each e-block starts with code to generate a prelog and ends with code to generate a postlog. The prelog consists of the values of the variables in the USED set of the e-block, and the postlog consists of the values of the variables in the DEFINED set. The PPD compiler also computes *GUSED* and *GDEFINED* sets for debugging purposes. GUSED set of an e-block is the set of variables that might be read-accessed by the e-block and by subroutines (transitively) called in the e-block. GDEFINED set is defined similarly.

The only condition for several consecutive lines of code to form an e-block is that the entry point for an e-block must be well defined. The entry point is where the prelog is made. The postlog is made at the exit point where the control is transferred out of an e-block.

The number of e-blocks that we construct from a given program is crucial to the performance of the system during the execution and debugging phases. While the number of logging points should be small enough so as not to introduce an unacceptable performance degradation during the execution phase, it should also be large enough so as not to introduce unacceptable time delay in reproducing traces during the debugging phase.

PPD compiler constructs an e-block out of each subroutine and an e-block out of a loop. Even though the size of a loop may be small, the execution time for these components may be long and may introduce unacceptable time delay in reproducing the traces. The PPD compiler constructs e-blocks from the loops so that the debugging phase can proceed without excessive time spent in re-executing the loops.

Small and frequently called subroutines can also be a problem. If we make an e-block out of each small subroutine, the amount of logging done during the execution phase may be large enough to introduce unacceptable performance degradation. To avoid this problem, PPD compiler provides an option to not make e-blocks out of the subroutines that correspond to leaf nodes in the call graph. In this case, the direct ancestor subroutines of these leaf subroutines *inherit* the USED sets and the DEFINED sets of the leaf subroutines, and perform the logging for the descendent subroutines.

3.2. Shared Variables

We use the log to generate detailed traces of events at debug time. However, for parallel programs with shared variables, the prelogs and postlogs described so far are not enough to produce the same debug-time traces as the traces that might have been generated during the execution phase. We need to save more run time information to ensure the reproducibility of parallel programs. Such additional information usually includes the values of the shared variables. We identify the additional information that we have to generate and where in the program we have to generate that information using the *simplified static graph* [4, 9] built at compile time. The simplified static graph is a subset of static graph that abstracts out everything except for the potential interactions between processes. There is one simplified static graph for each subroutine in the program.

Once we build the simplified static graphs of a program, we compute the *synchronization units* of the program. The synchronization unit roughly corresponds to the set of basic blocks that might be executed between two synchronization operations such as P and V semaphore operations. We apply interprocedural analysis in computing the synchronization units of a program [4]. Thus, each synchronization operation is associated with two synchronization units: one starting from that synchronization operation and the other terminating at that synchronization operation. PPD compiler generates code after each synchronization operation to produce an additional log entry for the synchronization unit starting from that synchronization operation. The log entry consists of the values of the shared variables read-accessed by the basic blocks of the synchronization unit. PPD compiler also generates code immediately before each synchronization operation to produce an additional log entry for the synchronization unit terminating at that synchronization operation. The log entry consists of the bit vector of the basic blocks of the synchronization unit. This trace entry is used to detect data races during execution.

4. Multi-Phase Compilation

The PPD compiler consists of four phases as shown in Figure 4.1. During the first phase, the compiler generates assembly code with labels in it to uniquely identify the places where future modifications might be needed. The decisions on logging code optimization are made during the second phase. Modification of the assembly code files are made during the third phase using the information generated during the second phase. These assembly code files are assembled and linked together during the fourth phase to yield the object code and the emulation package (described in Section 2). Piecewise data structures generated during the first three phases are also merged into global data structures during this fourth phase.

The first phase is a single module phase. During this phase, the compiler generates two assembly code files (one to be executed during run-time and the other to be executed during debug-time) from each source module. The compiler also generates local static program dependence graphs and a local program database for each module (file) during this phase; these local information files lack any interprocedural information.

The second phase is an inter-module analysis phase. During this phase, the compiler does interprocedural and shared-variable analysis of the program, using the static graphs and the program databases generated during the first phase. Based upon the results of these analyses, the compiler generates global information such as global static program dependence graphs, that has the interprocedural information. It also makes decisions on log optimization and on logging for shared variables.

The third phase is another single module phase, during which the compiler modifies each assembly code file generated during the first phase using information from the second phase.

The fourth phase is an inter-module merge phase, during which the assembly files are assembled and linked into the object code and the emulation package. Also pieces of global information generated during the second phase are merged into one global program database during this phase. We describe each phase in more detail in the following sections.

4.1. First Phase (Single Module Phase I)

Figure 4.2 shows the first phase of compilation for two example source modules (''x.c'' and ''y.c''). During this phase, the compiler generates two assembly files for each source module: one for the object code and the other for the emulation package. It also generates, for each source module, a local static graph file and a local program database file.

In generating the assembly files, the compiler does not optimize the logging code; it constructs one e-block out of each subroutine, and one e-block out of each loop. For nested loops, it currently constructs only one e-block from the outermost loop. Each e-block in the object-code assembly files starts with code to generate a prelog entry and ends with code to generate a postlog entry. However, these assembly files do not have logging code for the shared variables accessed in each synchronization unit. Logging for shared variables is determined during the second phase, and logging code is inserted into the assembly code files during the third phase.

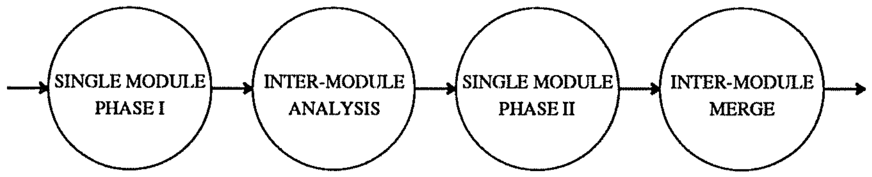


Figure 4.1. Overview of Four Phases of Compiling

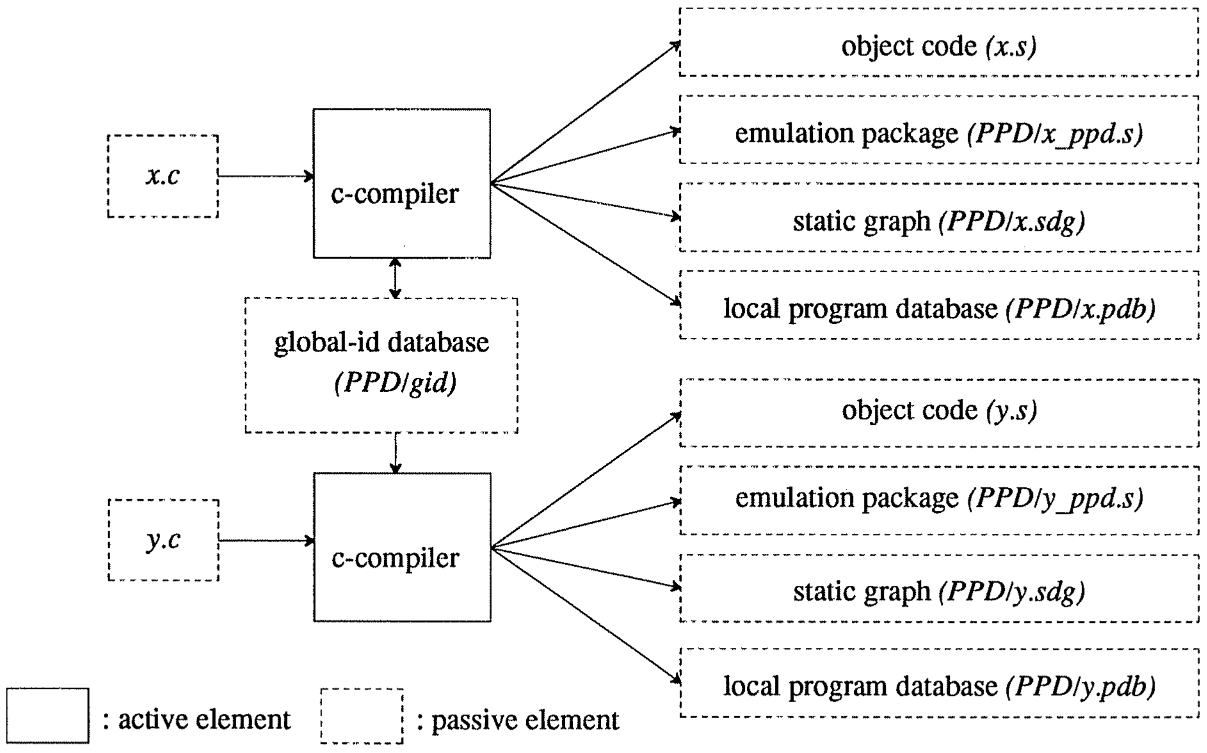

x.c
c-compiler
global-id database (PPD/gid)
y.c
c-compiler
object code (x.s)
emulation package (PPD/x_ppd.s)
static graph (PPD/x.sdg)
local program database (PPD/x.pdb)
object code (y.s)
emulation package (PPD/y_ppd.s)
static graph (PPD/y.sdg)
local program database (PPD/y.pdb)
: active element
: passive element

Figure 4.2. Single Module Phase (I)

To ease the modification of the assembly files, the compiler inserts unique labels at places in the assembly files where logging for shared variables might become necessary. Unique labels are also inserted into the assembly files to delimit the code related to the construction of e-blocks. These labels are used, during the second and third phases, to identify the places where the logging code should be optimized. A more detailed description of the modifications is given in later sections.

The static graphs generated during the first phase lack any interprocedural information. The program database contains information about symbols declared in each module. It also contains the USED and DEFINED set information for each e-block. Symbols such as variables or subroutine names are assigned unique identifiers during this phase. Such identifiers, not symbolic names, are used throughout the compile phases in identifying each symbol. The local program database also contains information to map the identifiers into symbolic names. In compiling several modules of a program, the compiler maintains the same identifier for a global name by using the *global-id database*. Whenever a new global symbolic name is met in compiling a module, the compiler consults with the global-id database to see if the global name is already assigned an identifier, and uses it if already assigned. If the name has not been yet assigned an identifier, the compiler assigns a new identifier to the global symbolic name and registers it in the global-id database.

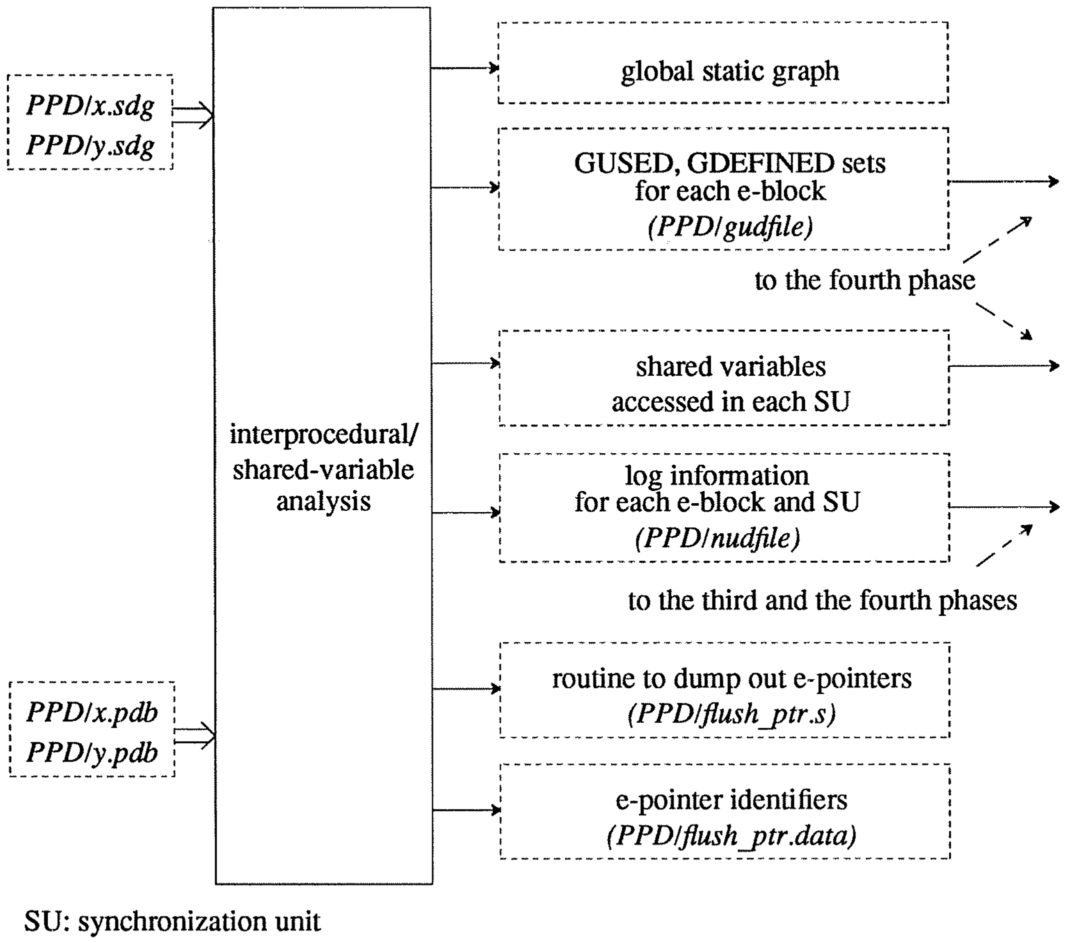

PPD/x.sdg
PPD/y.sdg
PPD/x.pdb
PPD/y.pdb
interprocedural/ shared-variable analysis
global static graph
GUSED, GDEFINED sets for each e-block (PPD/gudfile)
to the fourth phase
shared variables accessed in each SU
log information for each e-block and SU (PPD/nudfile)
to the third and the fourth phases
routine to dump out e-pointers (PPD/flush_ptr.s)
e-pointer identifiers (PPD/flush_ptr.data)
SU: synchronization unit

Figure 4.3. Inter-Module Analysis Phase

4.2. Second Phase (Inter-Module Analysis Phase)

The second phase, shown in Figure 4.3, is the interprocedural and shared-variable analysis phase. During this phase, the compiler builds the static call graph of the program, and computes the global used and defined sets of each e-block. The static call graph built by PPD is unique in that there is one node for each e-block; the nodes in the graph correspond not only to the subroutine call sites in the program but also to loops that constitute e-blocks. During this phase, the compiler also builds the global static graphs using the static call graph, the local static graphs, and the GUSED and GDEFINED sets.

The compiler decides how to optimize the logging code using the call graph and the GUSED and GDEFINED sets. The compiler first determines *non-eblock* subroutines — subroutines that will not form e-blocks. Subroutines corresponding to the leaf nodes in the static call graph are typically in this class. However, a subroutine that either contains a loop or contains accesses to a static variable (in the C Language) always forms an e-block. The compiler next identifies the *parent* e-blocks that call those non-eblock subroutines; these parent e-blocks will do the logging for the non-eblock subroutines. Last, the compiler decides what changes need to be made to the logging code of the parent e-block. The information about the changes to the logging code is recorded in file *PPD/nudfile*. PPD/nudfile is used in the later phases.

The compiler also computes the synchronization units and the sets of shared variables read in each of them. This computation uses the static graphs and local program databases. The compiler then generates logging code for shared variables and identifies the places in the assembly code where the logging code should be inserted. The resulting information is also recorded in PPD/nudfile.

Postlogs generated by the same e-block are linked in a list [3]. ''E-pointers'' is an array that contains pointers to the last log entry made by each e-block and is updated during program execution. The e-pointer array needs to be written to a file when the execution terminates. The compiler generates an assembly file that will write the e-pointer array to the log file when the program terminates. *PPD/flush_ptr.s* is the assembly file that contains the code to write out the e-pointers. This file will be compiled and linked with the object code during the fourth phase. *PPD/flush_ptr.data* is the file containing the list of e-pointers written by PPD/flush_ptr.s. This file is used by the Controller, when debugging is initiated, to match each e-pointer value with the corresponding e-block.

4.3. Third Phase (Single Module Phase II)

During the third phase, shown in Figure 4.4, the compiler makes changes to the object code and the emulation package assembly files generated during the first phase. The changes to the object code files are related either to the construction of e-blocks or to generating log entries for shared variables. First, the compiler deletes logging code that is no longer needed, such as

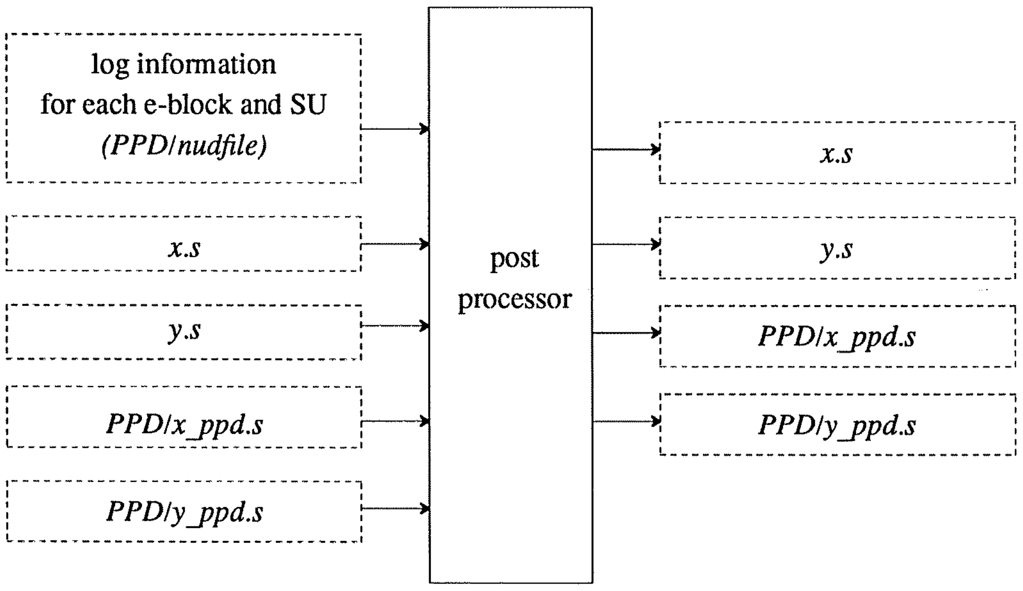
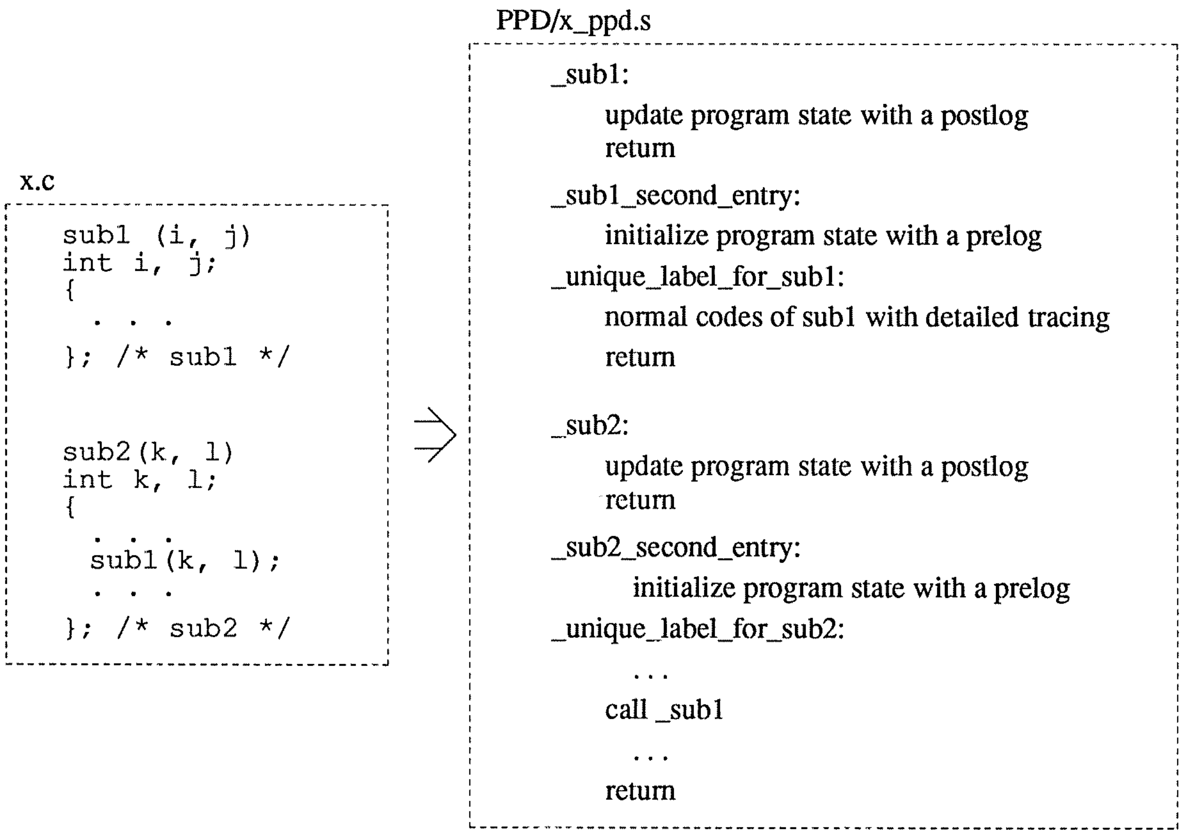


Figure 4.4. Single Module Phase (II)

code to generate prelogs and postlogs for non-eblock subroutines. Second, the compiler identifies and modifies the logging code of the parent e-blocks that will do logging for those non-eblock subroutines. Finally, the compiler identifies the places where additional logging for shared variables is needed and inserts the proper logging code at these places.

The compiler also modifies the emulation package for the non-eblock subroutines. Figure 4.5 shows two example subroutines and their corresponding emulation package e-block code. Entry point ''_sub1'' is used when subroutine ''sub1'' is called from other e-blocks. Upon entering ''_sub1'', the emulation package updates the program state with the corresponding postlog generated by ''sub1'' during run time. The program control then returns to the calling e-block and continues generating detailed traces of the calling e-block. Entry point ''_sub1_second_entry'' is used by the Controller to generate detailed traces for ''sub1''. If the compiler decides that ''sub1'' will not be an e-block, the *post processor* (see Figure 4.4) deletes the code between labels ''_sub1'' and ''_unique_label_for_sub1''. The effect is that the normal code of ''sub1'' will execute and generate detailed traces when entry point ''_sub1'' is entered from other e-blocks during debug time.

The post processor scans each assembly code file looking for the unique labels; such labels are planted into the assembly code files during the first phase. Whenever one of these



x.c

```
sub1 (i, j)
int i, j;
{
  . . .
}; /* sub1 */


sub2(k, l)
int k, l;
{
  . . .
  sub1(k, l);
  . . .
}; /* sub2 */
```

PPD/x_ppd.s

```
_sub1:
    update program state with a postlog
    return
_sub1_second_entry:
    initialize program state with a prelog
_unique_label_for_sub1:
    normal codes of sub1 with detailed tracing
    return

_sub2:
    update program state with a postlog
    return
_sub2_second_entry:
      initialize program state with a prelog
_unique_label_for_sub2:
      . . .
    call _sub1
      . . .
    return
```

Figure 4.5. Example Emulation Package E-blocks

labels is read, the post processor consults with the table built from PPD/nudfile to see if any modifications to the assembly file are necessary. The time complexity of this process is roughly proportional to the size of the assembly files.

4.4. Fourth Phase (Inter-Module Merge Phase)

The fourth phase, shown in Figure 4.6, is the inter-module merge phase. During this phase, the assembly files, generated during the first phase and modified during the third phase, are compiled into two executables: one to be executed at run-time (the object code) and the other to be executed at debug-time (the emulation package). Also, various information generated during the previous phases is merged into a global program data base to be used by the Controller at debug-time. File ''controller.o'' contains the Controller routines that control the debug time execution of the emulation package.

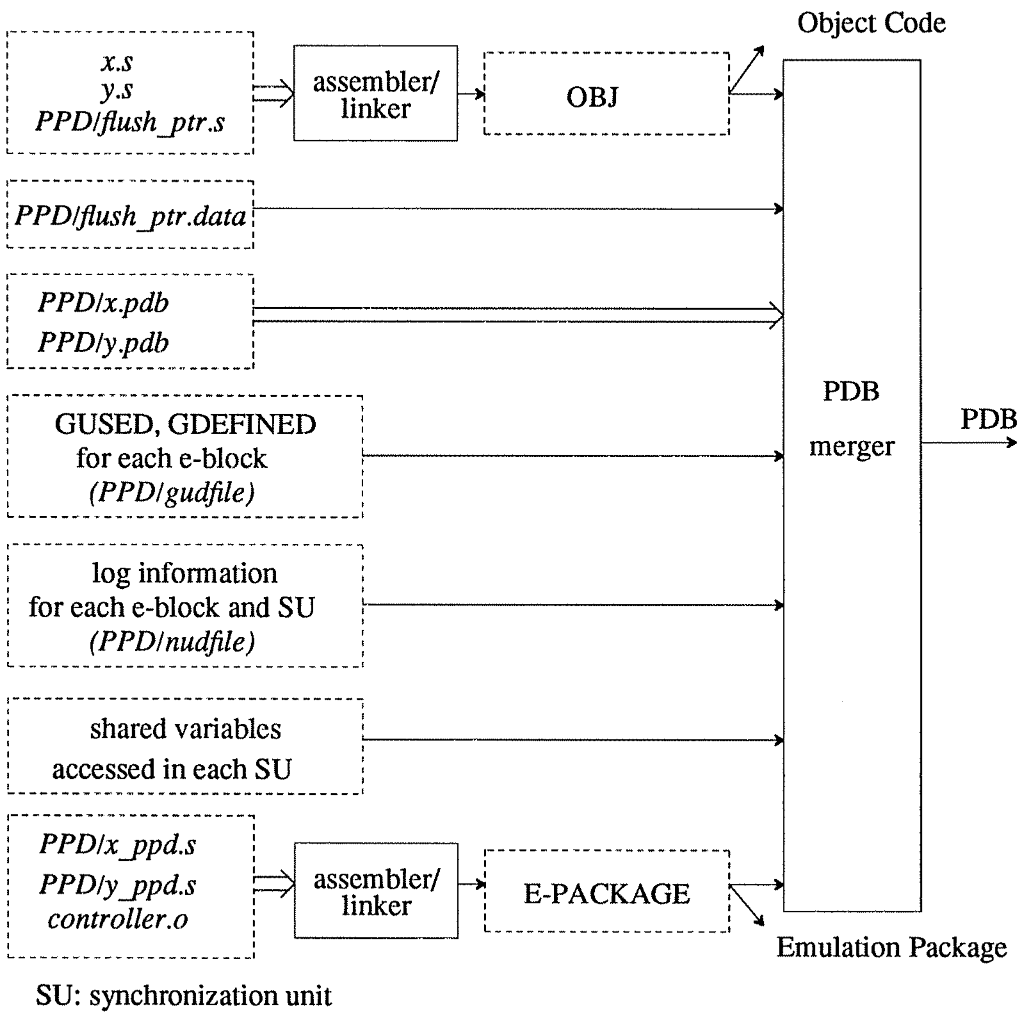


Figure 4.6. Inter-Module Merge Phase

5. Performance Measurements

This section presents measurements of the overhead caused by PPD on compile time, and on the size and execution time of application programs. We compare the performance of the PPD compiler with that of the Sequent Symmetry C compiler in the following categories: compilation time, size of the object code, and execution time of the object code. We also present the measurement of execution-time trace size. Figures 5.1–5.6 summarize the results of the comparison. We also discuss the trade-off between execution-time efficiency and debug-time efficiency.

In this paper, we present measurement results of four of our test programs: MATRIX, SH_PATH_1, SH_PATH_2, and CLASS. MATRIX multiplies two square matrices of integers into a third matrix. The size of each matrix, for our tests, is 100 by 100. MATRIX uses a subroutine to multiply pairs of scalar elements of the two matrices. The subroutine does not contain a loop or any accesses to static variables, making it a target of log optimization.

SH_PATH_1 computes the shortest paths from a city to 99 other cities using an algorithm described by Horowitz and Sahni [6]. SH_PATH_2 is the same as SH_PATH_1 except that it computes the shortest paths from all of the 100 cities to all of the other cities. CLASS is a program that emulates course registration for students, such as registering for courses and dropping courses. We obtained the test results of CLASS by running it with an input file containing 130 such registration activities. CLASS also can be run as an interactive program.

The CLASS test program consists of 5 separately compiled modules (source files). Each of the other programs consists of a single module. SH_PATH_1 and SH_PATH_2 each

		Sequent Compiler	PPD compiler w/o log optimization (overhead in %)	PPD compiler w/ log optimization (overhead in %)
MATRIX	CPU	1.4	5.4 (286%)	5.9 (321%)
	Elapsed	1.8	6.8 (278%)	8.4 (367%)
SH_PATH_1	CPU	2.0	7.0 (250%)	7.0 (250%)
	Elapsed	3.0	8.1 (170%)	8.1 (170%)
SH_PATH_2	CPU	1.7	6.8 (300%)	6.8 (300%)
	Elapsed	2.3	7.9 (243%)	7.9 (243%)
CLASS	CPU	5.1	15.9 (212%)	18.1 (255%)
	Elapsed	5.8	18.7 (222%)	21.7 (274%)

Figure 5.1. Compilation Time Measurements (time in seconds)

contain a subroutine that actually computes the shortest path. This subroutine is called 10 times by the main procedures. Thus the actual execution times of these two programs are approximately 1/10 of the times in Figure 5.4. Some test programs show no performance difference with log optimization; all subroutines in them have loops or accesses to static variables.

5.1. Compilation Time

Figure 5.1 summarizes the measurements of complete (compiling to linking) compilation times. CPU time is the sum of user time and system time. In general, the PPD compiler takes less than four times as long as the Sequent compiler. To compare the compilation time of the two compilers in more detail, we measured the times spent in each step of compiling program MATRIX, as shown in Figure 5.2. For the Sequent compiler, we divided the compilation into two steps: The first step (row 1) is to generate an assembly file from the source file and is done by the C compiler; this step corresponds to the first phase of the PPD compiler (row 4). The second step (row 2) is to generate an executable file from the assembly file and is done by the assembler and the linker; this step corresponds to the fourth phase of the PPD compiler (row 7).

The first phase of the PPD compiler takes about twice as long as that of the Sequent compiler, which is caused by the time spent on dependence analysis and in generating three additional files: one assembly file for the emulation package, a static dependence graph file, and a local program database file. The fourth phase of the PPD compiler takes longer than that of the

	compile phase	CPU	ELAPSED
Sequent Compiler	(1) ccom	0.6	0.8
	(2) assembler and linker	0.7	0.9
	(3) `total`	`1.3`	`1.7`
PPD Compiler with log optimization	(4) first phase	1.3	2.0
	(5) second phase	0.3	0.5
	(6) third phase	0.5	0.8
	(7.1) *object code assemble & link*	*1.3*	*1.4*
	(7.2) *emulation package assemble & link*	*1.3*	*1.5*
	(7.3) *global database*	*1.2*	*1.2*
	(7) fourth phase	3.8	4.1
	(8) `total`	`5.9`	`7.4`

Figure 5.2. Details of Compiling Time for Program MATRIX (time in seconds)

Sequent compiler, which can be accounted for by the time spent in generating two executables (object code and emulation package) and in merging piecewise data structures into large global data structures.

The fourth phase of the PPD compiler consists of three sub-phases: the first phase generates the object code (row 7.1), the second phase generates the emulation package (row 7.2), and the third phase merges all of the piecewise data structures into a global program database (row 7.3). Figure 5.2 shows that the times spent in each of the three sub-phases are roughly proportional to the sizes of the executable files that they produce; the sizes of the files are given in Figure 5.3.

A limitation of the current version of the PPD compiler is that it repeats in phases 2−4 whenever one of the source modules is modified. Thus, the PPD compiler shows relatively large overhead when a source module of multiple-module programs needs to be re-compiled. A detailed measurement results of compiler overhead for this case is given in [4]. We are currently working on *incremental semantic analysis* methods, which can identify and isolate the portion of semantic information to be updated due to changes in some part of the program.

		MATRIX	SH_PATH_1	SH_PATH_2	CLASS
Source Code		967	1691	1683	5488
Sequent Compiler Object Code		18355	20668	20667	29574
PPD Object Code (overhead)	without log-optimization	30556 (66%)	31574 (52%)	31323 (51%)	47809 (61%)
	with log-optimization	30520 (66%)	31574 (52%)	31323 (51%)	43621 (47%)
PPD Emulation Package (overhead)	without log-optimization	39875 (117%)	44994 (118%)	44484 (115%)	58688 (98%)
	with log-optimization	39933 (118%)	44994 (118%)	44484 (115%)	58532 (98%)
Program Database		2287	3421	3241	7432
Static Graph		1408	3265	3221	7410

Figure 5.3. File Sizes
(sizes in bytes)

5.2. File Sizes

Figure 5.3 shows the source and object code sizes of the programs tested. It also shows the size of the program database and static graph files. The size of the executables generated by the PPD compiler are 47 – 66% larger than the executable files generated by the Sequent compiler. In general, small programs have large proportional size increases, because the PPD compiler generates additional code (of fixed length) for procedure `main` to initialize the logging routines. MATRIX has the smallest original code size, yet it has the largest proportional increase in size (66%).

Programs with small subroutines have potentially large increases in object-code size due to the additional code to generate a prelog entry and a postlog entry for each subroutine call. However, log optimization will often reduce the logging overhead for these programs. CLASS has several small subroutines, and has a relatively large proportional size increase (61%) before log optimization. However, it has a smaller size increase (47%) after log optimization.

Emulation packages also have large increases in size. Such increases in size are expected, because emulation-package routines generate a trace record for each assignment statement.

5.3. Execution Time

The goal of the PPD design is to minimize execution time overhead without unduly burdening the other phases of program execution. Figure 5.4 shows the execution-time overhead of the tested programs. It shows that the execution time overheads range 0 – 330% for none log-optimized object code, and range 0 – 75% for log-optimized ones. We obtained the test results of CLASS by running it with an input file containing 130 registration activities such as registering for a course or dropping from a course.

MATRIX is the biggest winner of log optimization. The execution-time overhead of MATRIX is reduced by over 300% (from 330.7% to 7.9%) with log optimization. MATRIX has a subroutine that is called one million (100 by 100 by 100) times by another subroutine. Without log optimization, each call to this subroutine generates a prelog-postlog pair, resulting in large execution-time overhead (due to the one million prelog-postlog pairs). However, this subroutine does not have a loop or accesses to static variables; with log optimization, this subroutine becomes a non-eblock subroutine and the caller becomes the parent e-block. The non-eblock subroutine does not generate log entries, yielding a much smaller execution time. Accordingly, log optimization also causes MATRIX to have a large reduction in the size of execution-time traces.

Log optimization might actually produce a higher execution-time overhead if the non-eblock subroutine is never invoked due to conditional statements in the program; parent e-blocks of these non-eblock subroutines may generate additional log information for the non-eblock subroutines that are never invoked. However, we expect that such cases of losing by log optimization should be rare.

We also see that dumping out an entire array (for a log entry) at the beginning or at the end of a loop is inexpensive in terms of execution time overhead if most of the array elements are actually accessed in the loop. Such is the case with program SH_PATH_2. However, if only a fraction of the array elements are accessed in a loop, dumping out an entire array can be

		Sequent Compiler	PPD compiler w/o log optimization (overhead in %)	PPD compiler w/ log optimization (overhead in %)
MATRIX	CPU	12.7	52.5 (313.4%)	13.4 (5.5%)
	Elapsed	12.7	54.7 (330.7%)	13.7 (7.9%)
SH_PATH_1	CPU	1.1	1.8 (63.6%)	1.8 (63.6%)
	Elapsed	1.3	2.2 (69.2%)	2.2 (69.2%)
SH_PATH_2	CPU	107.0	105.5 (- 2.4%)	105.5 (- 2.4%)
	Elapsed	107.0	107.3 (2.8%)	107.3 (2.8%)
CLASS	CPU	0.3	0.4 (33.3%)	0.4 (33.3%)
	Elapsed	0.4	0.7 (75.0%)	0.7 (75.0%)

Figure 5.4. Execution Time Measurements (time in seconds)

expensive, as seen in test program SH_PATH_1. However, by using a more sophisticated analysis of dependences for complex data objects [4], we should be able to make the compiler smart enough to generate a log entry containing the particular row of the matrix that is actually accessed instead of the entire matrix.

Array logging can also cause some interesting performance anomalies. Notice that test program SH_PATH_2 shows a slight improvement in CPU time (the sum of user and system time) with the code generated by the PPD compiler. The PPD compiler generates logging code immediately before the loop that accesses a large array; the logging code accesses the entire array. This extra access seems to affect the paging behavior (possibly at the architecture level) of the program, resulting in less execution time. We are currently investigating this anomaly.

As mentioned before, CLASS can also run as an interactive program. While there is a 33% increase in CPU time and a 75% increase in elapsed time when CLASS ran using an input file, there was no noticeable difference in the response times when CLASS ran interactively. The relatively high execution-time overhead of program CLASS can be also explained by the fact that CLASS contains several loops in which only a fraction of an array is actually accessed.

5.4. Execution-Time Trace Size

Figure 5.5 shows the sizes of execution-time traces generated by the test programs. As described before, program MATRIX has a substantial reduction in trace size caused by the log optimization. Program CLASS has a slightly larger trace size with log optimization because of the reason that were previously described.

	without log optimization	with log optimization
MATRIX	40120221	120217
SH_PATH_1	825517	825517
SH_PATH_2	417129	417129
CLASS	104508	104892

Figure 5.5. Execution-Time Trace Size Measurements (sizes in bytes)

5.5. Trade-Off between Run Time and Debug Time

As described in Section 3, there is a trade-off between efficiency during execution and response time during debugging: If we construct an e-block in favor of the execution phase, debugging phase performance will suffer. On the other hand, if we construct an e-block in favor of the debugging phase, execution phase performance will suffer. In this section, we present initial results for the cost of debug time re-execution. The results are still quite limited, and we are currently performing a more set of extensive experiments.

Figure 5.6 shows the re-execution times and debug-time trace sizes of various e-blocks from our test programs. The e-block from MATRIX is made of a triply nested loop. By constructing a single e-block out of the triply nested loop of MATRIX, we were able to reduce the execution phase overhead, but with a large debug-time overhead: 166 seconds in re-execution time and 57.76 Mbytes of debug-time trace, generating more than one million detailed trace records. For a comparison, the execution time of MATRIX itself is 13 seconds, and its

	Re-execution Time		Debug-Time Trace Size	Elapsed-Time Overhead (†)
	CPU	ELAPSED		
e-block 1 (MATRIX)	160.5	165.5	57.76 Mbytes	7.9%
e-block 2 (SH_PATH_1)	3.8	4.8	1.24 Mbytes	69.2%
e-block 3 (SH_PATH_2)	> 364.8	> 422.8	> 117.79 Mbytes	2.8%

† (from Figure 5.4)

Figure 5.6. Re-execution Times and Trace Sizes (sizes in bytes)

execution-time trace size is 0.12 Mbytes (with log optimization). Note that we would only re-execute the log (and incur this cost) if we were interested in the details of a dependence within the loop.

The e-block from SH_PATH_1 is constructed out of a singly-nested loop that computes the shortest paths from one city to the 99 other cities, while the e-block of SH_PATH_2 is constructed out of a doubly nested loop that computes the shortest paths from 100 cities to all the other cities. Re-execution for the e-block from SH_PATH_1 took about 5 seconds with 1.24 Mbytes of trace. Re-execution of the e-block from SH_PATH_2 terminated because of insufficient file space for the detailed traces. At that time the e-block from SH_PATH_2 had re-executed for more than 7 minutes with more than 117.79 Mbytes of trace. These two results suggest that it might sometimes be better to construct more than one e-block out of a nested loop.

The additional e-blocks could nest, potentially forming an e-block around each of the nested loops. Alternatively, it is possible to divide a section of code into contiguous e-blocks. In this case, we would only need to re-execute the e-blocks that potentially contain the dependences in which we are interested. It may also be possible to decide dynamically, at execution time, whether or not to generate logs for an e-block. This dynamic decision could be based on the amount of time already spent in a loop. The decision of how to form e-blocks affects only program performance, so we are free to change these decisions without affecting the logic of the program. Of course, there is additional overhead involved in making these dynamic decisions, so we need to experiment to see if this mechanism would indeed be beneficial.

5.6. Discussion of Measurements

In this section, we have provided performance measurements of the various parts of PPD. The measurements show the expected increases in compilation time for the PPD compiler. Also, the measurements show that we need to take further advantage of separate compilation. The second phase (intermodule and shared variable analysis) of the PPD compiler does not need to be completely repeated. We are currently researching techniques to incrementally apply changes from static graphs of individual files to the global static graph. Similar techniques must be developed for the other global data structures.

The increase in execution time from generating the logs varies quite a bit for the various test programs ($0-75\%$). However, the larger increases in execution time come from test programs that access only parts of arrays in loops. With a more sophisticated dependence analysis for complex data objects [4], we expect substantial improvements to be possible for such programs.

Execution-time trace sizes are generally small (less than 1 Mbytes in all cases). However, the measurements show that we need more experiments to better understand the balance between the trace size during execution and the response time during debugging.

6. Conclusion

In this paper, we have described the code-generation techniques for separate compilation used in the implementation of the PPD debugger. We measured the overhead caused by PPD on compile time, and on the size and execution time of application programs. Not surprisingly, PPD compile time, with the addition of the emulation package, static graph, and program data-base, is several times slower than the standard system compiler. But these times still seem to be reasonable (i.e., they will not substantially impede the progress of the programmer). We believe that the programmer's added efficiency in finding bugs will more than compensate for the increment in compiling costs. The modified object files and auxiliary files together seem to be only 2–2.5 times the size of the object files from the standard system compiler.

We also presented the measurement of execution-time trace size, and discussed the trade-off between execution-time efficiency and debug-time efficiency. Naive tracing can generate large log files, but with the addition of some basic optimizations, the size of the log files can be quite reasonable. These optimizations must be traded-off with the cost of generating the detailed tracing during the interactive part of debugging.

In general, the performance measurements of PPD described in this section have demonstrated the feasibility of our ideas and directions for debugging parallel programs. However, PPD is a complex system and the studies in the paper are only preliminary. We need more experiments and research to better balance the trace size during execution and the response time during debugging.

The test programs used in the performance measurements of PPD are, in general, small in size. However, we think the results obtained with these program will scale proportionally to larger programs. As features are added to our debugging system, we are extending the types and sizes of test programs that we are studying. As we gain experience, we should be able to better evaluate our optimizations and to design new ones.

Acknowledgements

This research was supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

References

1. Allen, R. and Kennedy, K., ``Automatic Translation of FORTRAN Programs to Vector Form,'' *ACM Transactions on Programming Languages and Systems* **9**(4) pp. 491-542 (October 1987).

2. Balzer, R. M., ``EXDAMS–EXtendable Debugging and Monitoring System,'' *Proc. of AFIPS Spring Joint Computer Conf.* **34** pp. 567-580 (1969).

3. Choi, J. D., Miller, B. P., and Netzer, R., ``Techniques for Debugging Parallel Programs with Flowback Analysis,'' *Computer Sciences Technical Report #786*, Univ. of Wisconsin–Madison, (August 1988).

4. Choi, J. D., ``Parallel Program Debugging with Flowback Analysis,'' *Ph.D. Thesis*, Univ. of Wisconsin–Madison, (August 1989).

5. Cooper, K., Kennedy, K., and Torczon, L., ''The Impact of Interprocedural Analysis and Optimization in the **R**^{n} Programming Environment,'' *ACM Trans. on Programming Languages and Systems* **8**(4) pp. 491-523 (October 1986).

6. Horowitz, E. and Sahni, S., *Fundamentals of Data Structures,* Computer Science Press (1983).

7. Kennedy, K., ''A Survey of Data-flow Analysis Techniques,'' *Program Flow Analysis: Theory and Applications, S. S. Muchnick and N. D. Jones, Eds.,* pp. 5-54 Prentice-Hall, Englewood Cliffs, N.J., (1981).

8. Kernighan, B. and Ritchie, D., *The C Programming Language,* Prentice Hall, Inc., Englewood Cliffs, N.J. (1978).

9. Miller, B. P. and Choi, J. D., ''A Mechanism for Efficient Debugging of Parallel Programs,'' *Procs. of the SIGPLAN Conf on Prog. Language Design and Implementation,* pp. 135-144 Atlanta, GA, (June 1988). Also appeared in the *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, June 1988.