Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs

John Mellor-Crummey

CRPC-TR92232
September 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs*

John Mellor-Crummey†
Center for Research on Parallel Computation
Rice University
Houston, TX 77251-1892
johnmc@rice.edu

Abstract

Data race detection strategies based on software run-time monitoring are notorious for dramatically inflating execution times of shared-memory parallel programs. Without significant reductions in the execution overhead incurred when using these techniques, there is little hope that they will be widely used. A promising approach to this problem is to apply compile-time analysis to identify variable references that need not be monitored at run time because they will never be involved in a data race. In this paper we describe ERASER*, a data race instrumentation tool that uses aggressive program analysis to prune the number of references to be monitored. To quantify the effectiveness of our analysis techniques, we compare the overhead of race detection with three levels of compile-time analysis ranging from little analysis to aggressive interprocedural analysis for a suite of test programs. For the programs tested, using interprocedural analysis and dependence analysis dramatically reduced the data race instrumentation added by* ERASER*, resulting in a corresponding reduction of run-time monitoring overhead.*

1 Introduction

In an execution of a shared-memory parallel program, a *data race* is said to exist when there are two or more accesses to the same shared variable, at least one access is a write, and the temporal order of the accesses is not guaranteed by the sequential program flow or program synchronization. In the presence of a data race, the program's execution behavior may depend on the temporal order of the accesses participating in the race. The result is that the program may exhibit erroneous behavior in some, but not necessarily all, executions.

Detecting data races in shared-memory parallel programs is a difficult problem. Strategies for detecting such race conditions can be generally classified as (1) static analysis — analysis of a program's text to determine when two references may refer to the same shared variable [1, 3, 4, 24], (2) post-mortem analysis — collection of a log of events that occur during a program's execution and post-processing the log to isolate data races [2, 8, 17, 19, 20], or (3) on-the-fly analysis — augmentation of a program to detect and report data races as they occur during its execution [11, 12, 15, 16, 21, 22, 23].

Static analysis techniques rely on classical program dependence analysis and an analysis of a program's concurrency structure to determine when two references may potentially be involved in a data race. Static techniques are inherently conservative, which often leads to reports of data races that could never occur during execution. Experience with static analysis tools has shown that the number of false positives reported using these techniques is too high for programmers to rely exclusively on static methods for isolating data races. Combining static analysis with symbolic execution offers hope for reducing reports of infeasible races [26].

Post-mortem techniques for detecting data races have the advantage that they can limit reports to feasible races. However, to guarantee that only feasible races are reported, exhaustive execution trace logs are necessary. The size of such execution logs can be a serious drawback for these methods; complete logs can be enormous for parallel programs that execute for more than a trivial amount of time. Several approaches to post-mortem analysis that use smaller trace logs are promising. Netzer has studied the capabilities and limitations of post-mortem analysis based on partial information [17].

*This paper appeared in *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, pages 129–139, May 1993. (Available as SIGPLAN NOTICES, 28(12), December 1993.)

†This work was supported in part by National Science Foundation Cooperative Agreement CCR-9120008.

A promising alternative to pure post-mortem analysis is a hybrid approach that uses abbreviated logs containing only synchronization information to compute guaranteed orderings [18]. Such orderings can be used in conjunction with subsequent static analysis or on-the-fly monitoring to report race conditions.

On-the-fly techniques maintain additional state information during a program's execution to determine when unordered, conflicting accesses to a shared variable have occurred. Like post-mortem techniques, on-the-fly techniques can limit their reports to feasible races. The principal drawback of on-the-fly techniques is that for parallel programs with complex patterns of synchronization, the worst-case space and time overhead of detecting races during a program's execution includes a term proportional VT, where V is the number of shared variables and T is the maximum logical concurrency in the program execution [11]. However, when the structure of logical concurrency is restricted to that achievable with closed, nested fork-join constructs (e.g., nested parallel loops), the worst-case asymptotic space overhead can be reduced to $O(VN)$, where N is the maximum dynamic nesting depth of parallel constructs, and the asymptotic time for testing if a particular access participates in a data race can be reduced to $O(N)$ [15]. These tighter bounds offer the promise of efficient on-the-fly detection of data races for this restricted class of programs.

Both on-the-fly and post-mortem techniques rely on monitoring all potentially conflicting accesses to shared variables during a program's execution. The run-time monitoring associated with both of these classes of schemes is notorious for dramatically inflating execution times of shared-memory parallel programs. Without significant reductions in the execution overhead incurred when using these techniques, there is little hope that they will be widely used. To reduce the dynamic monitoring overhead incurred when using either class of techniques, it is important to minimize the number of accesses that are monitored at run time.

This paper focuses on using compile-time analysis to identify variable references that need not be monitored at run time because they will never be involved in a data race. We describe ERASER, a data race instrumentation tool implemented as a component of the Rice University's ParaScope system, that uses several levels of static program analysis, ranging from little analysis to aggressive interprocedural analysis, to prune the number of references instrumented for run-time monitoring. To quantify the effectiveness of each style of analysis, we compare the instrumentation overhead resulting from each style of analysis for a suite of test programs. Overall, for the programs tested, using interprocedural analysis and dependence analysis dramatically reduced the data race instrumentation added by ERASER, resulting in a corresponding reduction of run-time monitoring overhead.

Section 2 briefly describes ParaScope and provides context for the implementation of ERASER. Sections 3.1–3.3 describe three approaches to data race instrumentation that rely on increasing amounts of program analysis to reduce the number of access checks added to a program. Section 4 describes some experimental results that compare the effectiveness of using different levels of analysis when performing instrumentation for run-time monitoring. Finally, section 5 briefly outlines how the analysis used by ERASER could be enhanced to further reduce the number of references that would be instrumented for run-time monitoring.

2 ParaScope

Most visibly, ParaScope is a suite of tools that support scientific programming in Fortran. Underneath, ParaScope provides an infrastructure for management, analysis, and transformation of programs written in a Fortran dialect with extensions for shared-memory parallelism. This infrastructure includes facilities that support a rich collection of program analyses including dependence analysis, control flow graph analysis, computation of static single assignment form, global value numbering, and a variety of interprocedural analyses. The ParaScope infrastructure has served as a platform for research on aggressive optimization of scientific codes for both scalar and shared-memory machines [7].

A key function of ParaScope is to support whole-program (interprocedural) analysis and optimization. To a large extent, support for whole-program analysis in ParaScope preserves the benefits of separate compilation of procedures by minimizing the number of times a procedure is examined. In general, interprocedural analysis and transformation in ParaScope uses the following 3-phase approach [7, 9, 13]:

1. **Local Analysis.** At the end of an editing session with one of the editors in the ParaScope toolset, summary information concerning all local interprocedural effects for each procedure in the edited module is calculated and stored.

This information includes details on call sites, formal parameters, scalar and array section uses and definitions, local constants, symbolics, loops and index variables. Since the initial summary information for each module does not depend on interprocedural effects, it only needs to be collected once when the module changes even if the module is used as part of several programs. (If a module is updated by something other than one of the editors in the ParaScope toolset, local analysis information is collected and saved the first time interprocedural analysis is requested after the module has changed.)

2. **Interprocedural Propagation.** The interprocedural analysis system gathers local summary information collected for each procedure and uses it to build a call graph. The analyzer then uses other components of the local summary information as the basis for computing solutions to interprocedural dataflow problems on the call graph.

3. **Code Generation.** ParaScope tools that consume interprocedural information perform transformations of procedures based on the results of interprocedural analysis.

In order to efficiently support whole-program analsis and transformation, it is important to minimize the cost of program changes. In an interprocedural system, even modules that have not been edited may require updates if they are indirectly affected by changes to some other module. ParaScope has served as a testbed for *recompilation analysis* which identifies modules that have not been affected by program changes and do not need to be updated [5, 10].

Currently, ParaScope computes interprocedural REF, MOD, and ALIAS; implementations are underway to incorporate computation of interprocedural CONSTANTS and a number of other important interprocedural problems, including interprocedural symbolic and regular section analysis of arrays.

3 Data Race Instrumentation

Internally, ParaScope represents program modules in the form of abstract syntax trees annotated with semantic information. A program transformation subsystem supports arbitrary transformation and augmentation of Fortran ASTs. This system serves as the framework for ERASER, our data race instrumentation tool. ERASER uses the transformation subsystem to augment Fortran ASTs by adding calls to run-time support routines that monitor synchronization events and variable references. As it is currently implemented, ERASER is intended to be used in conjunction with a run-time library that supports on-the-fly monitoring and reporting of data races in program executions. For this reason, ERASER also augments programs with definitions of auxiliary storage used to maintain access histories for shared variables potentially involved in data races. With relatively minor modifications, ERASER could be adjusted to omit these access history declarations to generate instrumented programs more suitable for use with a run-time library that collects reference and synchronization traces for post-mortem analysis.

ERASER and it's companion run-time library were designed to efficiently support detection of data races that arise in programs with closed, nested fork-join parallelism [15]. ERASER's implementation focuses on instrumentation of of programs with concurrency specified in the form of nested parallel loops. Support for heterogeneous parallelism that arises from Fortran parallel section constructs (analogous to the more familiar cobegin-coend construct of other parallel languages) could readily be added to ERASER if full support for heterogeneous parallelism was present in ParaScope's analysis infrastructure. Whereever the lack of support for heterogeneous parallelism is important, the term *parallel loop* will be used in place of the term *parallel construct* in order to remain faithful to the actual implementation.

To prepare a program for data race instrumentation, ERASER applies the following sequence of transformations to put the code in a canonical form:

- Transform logical IF statements into block IF statements so that instrumentation can be readily be added to the consequent as necessary.
- Transform each ELSEIF construct into an IF-THEN construct nested inside an ELSE construct. If any access in the ELSEIF guard needs instrumentation, the system must have a place to insert the instrumentation so that the access check gets executed *iff* the guard will be executed.
- Hoist all function invocations out of subscript expressions. Since subscript expressions will be duplicated into access check code, subscript expressions must be side-effect free.

- Move each statement label to a CONTINUE that precedes the statement. Since the system will insert data race instrumentation for a statement immediately before the statement, the system must ensure that it is impossible to reach a statement without executing its corresponding access check instrumentation.

Once the code is in a canonical form, the data race instrumentation can proceed. The data race instrumentation process consists of adding several different types of statements:

- concurrency bookkeeping — calls to race detection run-time library routines to indicate the creation or termination of a logical thread[1],
- access checks — calls to the race detection run-time library READCHECK or WRITECHECK operations that test if an access participates in a data race,
- access history declarations — each variable that may be involved in a data race is allocated storage for an access history in which information is stored about the threads that have accessed the variable, and
- access history initialization and finalization — access histories for all local variables must be initialized upon procedure entry and finalized before the procedure returns or halts.

The next three sections describe data race instrumentation strategies that rely on increasing levels of program analysis.

3.1 Basic Strategy

Without any sophisticated analysis, data race instrumentation must be very conservative. Each procedure must assume that it is called in the scope of a parallel construct. Therefore, inside a procedure references to its formal parameters (which are passed by reference in Fortran) and global variables must be instrumented since they could conflict with other accesses made in the context of an enclosing parallel construct. The system must also add access checks for any references to the procedure's local variables that occur inside the scope of a parallel construct in the procedure. This is necessary since without further analysis one cannot be certain that these references do not cause data races within the scope of some enclosing parallel construct.

[1] We use the term *thread* to denote the basic unit of concurrency (e.g., an interation of a parallel loop body).

Even if a procedure contains no parallel constructs, all local variables of the procedure are passed as actual arguments to user-defined procedures must have access history storage allocated in the scope in which the variables are declared and references to that storage must be passed to each called procedure since any called procedure could contain a parallel construct inside of which it references its arguments. Local variables not passed to called procedures need not be instrumented nor have access history storage allocated if no parallel constructs are present in the procedure in which the variables are declared.

Variable references passed to intrinsic functions require special handling. Intrinsic functions in Fortran read, but never modify their arguments. Since the bodies of instrinsic functions are not instrumented by ERASER, in some cases the system must add instrumentation at the point of call to reflect that the intrinsic reads its arguments. In particular, a READCHECK for a variable reference passed to an intrinsic must be added at the point of call if either of the following conditions are satisfied

- the variable is a formal parameter or a global variable, or
- the variable is a local to the procedure calling the intrinsic and the call to the instrinsic is made inside the scope of a parallel construct inside the procedure.

For each statement, the instrumentation system accumulates the set of variable references that need data race instrumentation. If multiple array element references in the same statement have the same sequence of subscript expressions, only one access check is needed for all of the references. This is true even if the references are a mix of *reads* and *writes* — in this case, a single `WRITECHECK` will suffice since any access that conflicts with a read will certainly conflict with a write.

3.2 Intraprocedural Strategy

To detect all data races, not all references to shared variables inside parallel loops need be instrumented. In particular, variable references to memory locations that are not accessed by more than one thread of control do not need data race instrumentation.

Data dependence analysis is a deep compile-time analysis of program variables and their subscripts to determine when two variable references may refer to the same memory location. Compile-time dependence analysis computes a conservative superset of

the dependences that may occur during a program's execution. In ParaScope, a dependence graph contains an edge for each data dependence, where each node in the graph represents a variable reference. A dependence edge between references R_1 and R_2 is *carried* by a loop if the access through R_1 in loop iteration i can potentially access the same memory location as the access through R_2 in loop iteration j, $i \neq j$. Dependences that are not carried by loops are said to be *loop independent*.

Three types of carried data dependences are important for data race instrumentation. A *true* dependence (also known as *flow* dependence) signifies that a memory location written during some loop iteration may be read in a later iteration. An *anti* dependence signifies that a memory location read during some loop iteration may be overwritten in a later iteration. Finally, an *output* dependence signifies that a memory location may be written during more than one loop iteration. When one of these types of data dependences is carried by a parallel loop, at run time two different loop iterations may perform concurrent, conflicting accesses to the same memory location resulting in a data race.

In the absence of procedure calls, all variable accesses that may be involved in a data race must be the endpoint of some data dependence that is carried by a parallel construct. In this case, it suffices to add access checks only for variable references that are endpoints of data dependences.[2]

When procedure calls are present, but no interprocedural information is available, conservative assumptions are necessary to ensure correctness. When building a dependence graph, conservative assumptions must be made about the side effects of each procedure call. In particular, the dependence analyzer must assume that each procedure call modifies each of its actual parameters (in fact, the analyzer must assume that any time a reference to an array element is passed to a procedure, the procedure modifies the whole array) and all global variables. To make sure data races involving accesses in different procedures will be detected under these circumstances, we fall back on the basic instrumentation strategy presented in the previous section. Each procedure must conservatively assume that it is invoked from inside a parallel construct; this requires access checks for references to global variables and the procedure's formal parameters in addition to access checks at dependence endpoints carried by parallel constructs within the procedure itself.

In comparison with the basic strategy presented in the previous section, the intraprocedural strategy has the potential for reducing instrumentation in programs by eliminating access checks for all references to each procedure's local variables other that those that are endpoints of data dependences carried by a parallel construct.

3.3 Interprocedural Strategy

In the instrumentation strategies presented thus far, conservative assumptions are made in the presence of procedures. At each call site, the system must assume that the called procedure modifies each of its actual parameters and all global variables. Furthermore, the system must assume that each procedure may be invoked from within a parallel construct.

These two assumptions lead the system to insert instrumentation conservatively. Interprocedural information can help the instrumentation system reduce the amount of data race instrumentation and its run-time overhead in two simple ways:

- If the system knows that a procedure is never called from within a parallel construct, no access checks for references to the procedure's formal parameters are necessary in the procedure except where a reference to a formal parameter is an endpoint of a data dependence carried by a parallel construct within the procedure.

- If the dependence analyzer has interprocedural summary information about the side-effects (MOD and REF with ALIAS information incorporated) of procedure calls in parallel constructs, it will not have to make the conservative assumption that all variables accessible to the procedure are modified. This can reduce the number of dependence endpoints, thus reducing instrumentation.

Additional improvements can be obtained in more subtle cases using the interprocedural analysis strategy described below. The description of the implementation strategy is presented for each of the three analysis phases in the ERASER corresponding to the framework described in section 2: local analysis, interprocedural propagation, and code instrumentation.

[2] ParaScope does not compute data dependences carried by parallel section constructs; thus, ERASER's instrumentation supports only parallel loops.

```
parallel loop i = 1, n
  call f(a[aindex[i]], b[i],  x[i])
end loop
...
parallel loop i = i, n
  call g(d[i], e[eindex[i]], y[i])
end loop

subroutine f(f1, f2, f3)
  call h(f1, f2, f3)
end

subroutine g(g1, g2, g3)
  call h(g1, g2, g3)
end
```

Figure 1: Different contexts have different instrumenation requirements.

Local Phase

As described in section 2, when a module is changed, initial information about a module is recorded for use during interprocedural propagation. Before support for data race instrumentation was envisioned in ParaScope, initial information recorded included a descriptor for each procedure specifying the names and types of formal parameters, initial MOD and REF information for formal parameters and common variables, call site descriptors including name of the invoked procedure (or procedure variable) and the actual arguments. To support data race instrumentation, this information was reorganized so that it is not summarized at the procedure level, but rather collected at the loop level. Also, the information was augmented to contain a description of the loop nesting structure and an indication of which loops are parallel. Loop-level information is important for data race instrumentation so that interprocedural analysis can determine which procedures are (possibly transitively) invoked from within the context of a parallel loop.

Interprocedural Phase

As was the case before support for data race instrumentation was envisioned as part of ParaScope, a call graph is constructed and interprocedural ALIAS, MOD, and REF summary information is computed for each procedure using the initial information collected during the local phase.

At this point, the interprocedural analyzer also computes a solution for the AFORMAL problem which offers a very crude approximation to array section analysis (which is not yet available in ParaScope). The AFORMAL set for a procedure indicates for each formal parameter of the procedure if it is ever referenced using array subscripting operations by the procedure or any of its decendants in the call graph. The solution of this interprocedural problem is useful for sharpening dependence analysis involving a procedure's side-effects. In particular, if an array element reference passed to a procedure is bound to a parameter not in the procedure's AFORMAL set, then the dependence analyzer knows that any MOD or REF side effect of the called procedure affects only the array element passed as an argument; thus, the dependence analyzer need not assume that the whole array had been modified or referenced by the procedure.

Next, a null *data race instrumentation set* is created for each procedure. After interprocedural analysis is complete, the data race instrumentation set for a procedure will indicate which formal parameters and global variables require access checks for references to them inside the procedure body.

The interprocedural analysis driver then invokes the dependence analyzer for each procedure using the interprocedural solutions for MOD, REF and AFORMAL to increase the precision of dependence information involving call site side effects. Using these interprocedural solutions, the dependence analyzer identifies when a data dependence involves side-effects of a call site. Such dependences indicate accesses made by the called procedure (or its descendants in the callgraph) that may be involved in data races; instrumentation will be needed for any access to that variable in the called procedure (or its descendants).

For each dependence endpoint at a call site (referring to an actual or global accessed as a side-effect of the call), the data race instrumentation set for the procedure is augmented to indicate that some context in which the procedure is called requires instrumentation for references to a particular formal parameter or global. When all of the call sites inside parallel loops in the program has been processed, the instrumentation sets are ready for dataflow propagation through the edges in the callgraph. Final values for the instrumentation sets result from forward dataflow propagation of all of the data race instrumentation sets along call site edges in the callgraph. At each call site, instrumentation requirements for the formals and globals of the callee are augmented so that they subsume any instrumentation requirements that the caller has for globals and variables passed as actuals at the call site. More precisely, during dataflow prop-

agation, the instrumentation requirements flowing to a procedure node from each of its incoming call site edges are unioned to achieve the final version of the data race instrumentation set for that procedure.

A brief, contrived example shown in figure 1 illustrates how context can impose different instrumentation requirements on a procedure's formal parameters. Assume that subroutine `h` accesses each of its formal parameters only as scalars and that it modifies its first two formal parameters, but only reads the third. Interprocedural MOD and AFORMAL sets will indicate that subroutines `f` and `g` modify their first two arguments (via their call to `h`). In the context of the first loop, the ParaScope dependence analyzer cannot prove that accesses to `a[aindex[i]]` are independent (the dependence analyzer does not track the values of indirection arrays, so it must conservatively assume that there are repeated values and report a loop-carried data dependence), but can prove that modifications to `b[i]` are independent (no carried dependence on the side-effect to `b[i]`). Since the third parameter to `f` is not in `f`'s MOD set, there is no loop-carried dependence involving this parameter. The context of this loop thus requires instrumentation inside `f` and `h` only for references to the routines' first formal parameter. In the second loop, the situation is reversed for the call to `g`: only accesses to its second formal parameter require instrumentation. After interprocedural dataflow propagation of the instrumentation sets, inside `h` references to its first and second formal parameters require instrumentation, but references to the third parameter do not.

For each procedure, its final data race instrumentation set describes which formal parameters and global variables require access checks inside the procedure body. However, with only the information computed thus far, each caller must conservatively assume that each actual argument that it passes to a called procedure requires access history storage. In the example shown in figure 1, there is no way for the loop calling `f` to know that the second parameter to `f` requires access history storage but that the third does not since these requirements are dictated from below by `h`.

Which variables require access histories allocated can be computed in a single backward dataflow pass over the callgraph. The initial value of the *storage allocation set* for each procedure is a copy of the procedure's final data race instrumentation set. During a backward dataflow pass, the storage allocation sets flow to each procedure from all the procedures it calls (along an edge for each call site inside the procedure). For each call site, only the variables known to the caller are propagated through the call site up to the caller. The sets propagated to a procedure node from each of its outgoing call site edges are unioned to achieve the final version of the storage allocation set for the procedure.

After applying this analysis to the program fragment shown in figure 1, the call sites for `f` and `g` will know that no access history storage is needed for `x` and `y` respectively. Furthermore, the call interface for each of the procedures needs to be expanded to include references to access history variables only for the parameters that actually require access history storage instead of for all variables (as would be the case in the basic and intraprocedural strategy).

After the storage allocation set is computed for each procedure, only the top-level program is aware of all of the common variables that must be expanded. A forward interprocedural dataflow pass is necessary to guarantee that each procedure has a consistent definition of which variables in each common block need to be augmented with access history storage. This third pass creates a *common allocation set* for each procedure.

Code Instrumentation

After all of the interprocedural analysis is complete, the data race instrumenter uses the information collected to instrument the Fortran AST for the program. Each reference that is an endpoint of a dependence carried by a parallel loop has a corresponding access check added. Also, each reference to a variable in a procedure's data race instrumentation has a corresponding access check added. For each local variable in the procedure's storage allocation set, access history storage is allocated, and calls to run-time support routines are added to initialize and finalize the locally-allocated access history storage upon entry and exit of the procedure, respectively. Common block definitions are expanded with access history storage added for each variable in the procedure's common allocation set. Actual argument lists are expanded at call sites to pass access history storage only for parameters in the callee's data race instrumentation set. Calls to concurrency bookkeeping routines are added only for parallel constructs that carry a data dependence. Thus, if a data race can never occur in the context of a parallel construct, no concurrency bookkeeping is performed at run time.

	static measures			dynamic measures		
	source lines	access checks read	access checks write	access checks read	access checks write	execution time
uninstrumented	569	0	0	0	0	36.8
basic	1073	70	17	111498737	15270102	598.3
intraprocedural	1065	68	17	100749809	15270102	548.8
interprocedural	697	4	7	23368976	15269825	251.4

Table 1: Data race instrumentation statistics for the search program.

	static measures				dynamic measures		
	source lines	instrumented loop nests	access checks read	access checks write	access checks read	access checks write	execution time
uninstrumented	1930	0	0	0	0	0	22.5
basic	3582	7	234	63	70085036	7295204	341.1
intraprocedural	3582	7	234	63	70085036	7295204	336.8
interprocedural	2251	3	20	24	17234673	4819904	103.9

Table 2: Data race instrumentation statistics for the buck program.

4 Experimental Results

In order to test the efficacy of the compile-time analysis strategies described in the previous section it is important to apply the analysis to some real programs. To date, we have carefully studied results with three shared-memory parallel programs.

The first program, *search*, implements a multi-directional direct search method for finding a local minimizer of an unconstrained minimization problem [25]. Search contains four parallel loop nests (each of which contain a call to the same evaluator function) surrounded by an outer serial loop that tests for convergence. The second program, *buck*, tests the adjointness of a routine that computes a one-dimensional seismic inversion (used for oil exploration) with its associated adjoint code. The code contains seven parallel loop nests, four of which contain calls to substantial procedures. The third program, *erlebacher* is a benchmark written by Thomas Eidson at NASA ICASE. It performs three dimensional tri-diagonal solves using alternating implicit direction integration. The version of erlebacher used for this study was was parallelized by Kathryn McKinley as part of her dissertation research at Rice; this version of erlebacher contains eighteen parallel loops, none of which contain procedure calls other than to intrinsics.

Table 1 contrasts static and dynamic statistics for the search program. The table shows measures for both the uninstrumented code and for code automatically instrumented by ERASER using the three different data race instrumentation strategies. The first column in the table shows source lines comparing the size of the original uninstrumented program versus the size with each style of data race instrumentation. The dramatic increase in source line count reflects the addition of access checks, concurrency bookkeeping calls, declarations for access history variables, calls to initialize and finalize each locally declared access history, as well as declarations and data statements that contain information that enables ERASER's associated data race run-time support library to pinpoint the location in the source code of each reference involved in a data race. The next two columns respectively indicate how many READCHECK and WRITECHECK calls were added to the program using each instrumentation strategy. Compared to the basic strategy, the interprocedural approach reduces the combined number of access checks added to the code by 87%. Since the remaining access checks are all in the computational kernel, the effective reduction of the dynamic access checks is not nearly as dramatic. In comparison to the basic strategy, the interprocedural strategy reduced the combined number of dynamic checks by 70%.

Access checks remaining in search after interprocedural analysis and data dependence analysis result from use of index arrays. As mentioned earlier, since the dependence analyzer does not track the values of array variables, it conservatively assumes that index arrays may contain replicated values which would imply a data dependence. Interestingly, search contains a pair of call sites that provide different contexts for a call to the same procedure. This concept was illustrated earlier in

	static measures			dynamic measures		
	source lines	access checks read	access checks write	access checks read	access checks write	execution time
uninstrumented	1234	0	0	0	0	21.9
basic	3265	281	80	14853692	3799360	104.5
intraprocedural	2996	240	71	13229914	3798784	95.8
interprocedural	1285	0	0	0	0	21.6

Table 3: Data race instrumentation statistics for the erlebacher program.

figure 1. By constructing a specialized version of the procedure for use in the different contexts (this process is known in the literature as procedure cloning), access checks to an array parameter only referenced by the called procedure could be eliminated for the clone invoked in the context in which there is no data dependence carried by the REF side effect of the parameter. This cloning would offer little run-time benefit however since the context that permits the clone without instrumentation is only executed during problem initialization.

Table 2 contrasts the same measures for the buck program. Dependence analysis alone was able to determine that there are no dependences carried by three parallel loops, each encapsulated in its own procedure. However, since each of the loops contains accesses to the arguments of the enclosing procedure, access checks are still necessary inside the parallel loops since nothing is known about the contexts in which the procedures containing the loops are called, and whether accesses to the arguments could cause data races. With interprocedural information, the instrumentation system is able to determine that none of the procedures containing a parallel loop is called from within another parallel loop; therefore, all instrumentation can be omitted from the aforementioned three parallel loops immediately. For a fourth parallel loop that contains a call to a procedure, all instrumentation also was eliminated because the analysis was able to determine that the side effects of the procedure did not result in any carried dependences. (No instrumentation was needed inside the procedure called from within the fourth parallel loop either.) The interprocedural approach reduces the the combined number of static access checks by 85% over both the basic and intraprocedural strategies. The interprocedural strategy reduced the number of dynamic checks 71% in comparision to the other approaches.

For buck, the remaining access checks are caused by the lack of array section analysis in ParaScope's dependence analyzer. The buck program is written in a modular style. For each of the parallel loops with procedure calls, with interprocedural array section side effect analysis it could be determined that the called procedures only modify a column of the arrays passed as parameters. With this information, the dependence analyzer could determine that the array columns passed to the called procedures are independent and data race instrumentation could be avoided.

Table 3 presents the measurements for the erlebacher program. Using dependence analysis alone, ERASER was able to determine that there were no intraprocedural dependences carried on any of the parallel loops. However, this resulted in the elimination of only a few of the access checks since each procedure must still assume it could be called from within a parallel loop and thus all procedures still required checks for accesses to their formals and globals. The combination of dependence analysis and interprocedural analysis was able to determine that no accesses would be involved in data races at run time and thus it was possible to eliminate access checking instrumentation. The careful reader will notice that the number of lines in the code processed by ERASER using interprocedural analysis differs from that of the original source even though no access checking instrumentation was added. This difference in source code lines is due to the code normalization process. The original program had many nested loops terminated with the same continue. During the code normalization phase of ERASER, each loop received its own continue statement.

All execution times reported in the last column of tables 1–3 are from sequential executions of the instrumented programs on a Sun[3] 4/490. Timings for the program executions correspond to the user time reported by the csh time command for each execution. While timings obtained in this manner will vary somewhat (some jitter can clearly be seen by comparing the timings reported), these times are accurate enough for the qualitative comparisons we want to draw here. Elsewhere we have

[3] Sun is a trademark of Sun Microsystems

shown that sequential executions suffice for detecting data races in programs with loop-based parallelism [15]. All programs were compiled with the Sun f77 compiler using -O optimization. Comparing raw execution times of the uninstrumented and instrumented code varieties shows the run-time overhead for on-the-fly monitoring to be relatively high. These numbers offer a conservative picture of the overhead of on-the-fly monitoring since the ParaScope data race run-time library is written in a modular style of C++ and has not been tuned for performance; for instance, the concurrency bookkeeping routines invoke "malloc" for dynamic memory allocation of each thread label rather than a tuned special-purpose allocator. For the search program, the execution overhead (computed as (instrumented execution time - uninstrumented execution time)/uninstrumented execution time) of the basic strategy was a factor of 15.5 whereas the interprocedural approach reduced this to a factor of 5.8. For the buck program, the execution overhead was a factor of 14 for the basic and intraprocedural strategies. The interprocedural strategy reduced the run-time overhead for race instrumentation of buck to a factor of 3.6. For erlebacher, the execution overhead using the basic instrumentation strategy was a factor of 3.8, while the interprocedural strategy eliminated the overhead entirely.

An interesting yardstick for putting some perspective on the overhead measured for data race detection when using the interprocedural instrumentation strategy is to compare the execution times of the resulting executables with times for uninstrumented executables compiled with the -g compiler flag. For the search program, the -g execution time was 138.7 seconds; using the interprocedural instrumentation strategy compiled with -O, data race detection costs only 78% more than the -g execution. For the buck program, the cost of a -g execution was 60.8 seconds; here the overhead of data race detection using the interprocedural instrumentation strategy compiled with -O is 198% more than the -g execution. For erlebacher, the comparison is pointless since the interprocedural instrumentation strategy eliminated all of the instrumentation.

5 Conclusions

Using dependence analysis and interprocedural analysis of scalar side effects as a basis for data race instrumentation, ERASER was able to reduce the dynamic counts of instrumentation operations monitoring for the presence of data races by 70–100% for the three programs tested. Construction of the ERASER prototype shows that such compile-time analysis for this purpose is practical and profitable. Further reductions in overhead are possible by increasing the sophistication of the analysis. In particular, with interprocedural array section analysis of procedure side effects, all instrumentation could be eliminated for the buck program. One must bear in mind though that the cases for which run-time monitoring is important are those in which compile-time analysis cannot prove the absence of data races in a program, as with the search program.

Other enhancements to ERASER's analysis that could provide reductions in instrumentation for programs other than those tested are forms of inter-statement analysis that could remove redundant access check operations within a procedure. (Currently, ERASER performs no inter-statement analysis.) Inter-statement analysis strategies that could reduce instrumentation requirements include using global value numbering, control flow graph domination, and analysis developed for recognizing reuse of array variables [6] to eliminate redundant access check operations.

Even with the impressive reductions in dynamic counts of monitoring operations that ERASER was able to achieve, monitoring overhead for automatic detection of data races ran as high as a factor of 5.8. Likely, a factor of two or more further reduction in the overhead could be obtained by carefully tuning the run-time library support routines. Compared to the alternative of tracking down data races by hand in program executions, the overhead of dynamic monitoring strategies such as the on-the-fly strategy supported by the run-time library tested here would likely seem to be acceptable for use in a testing phase by a frustrated user trying to track down the causes of indeterminate behavior of a program.

Acknowledgments

Robert Hood implemented a large part of the program transformation system upon which the data race instrumenter is built and most of the intraprocedural instrumentation system prototype.

References

[1] R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proc. of the 1986 Inter-*

national Conference on Parallel Processing, pages 164–170, Aug. 1986.

[2] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. of the 1987 International Conference on Parallel Processing*, pages 721–727, Aug. 1987.

[3] W. F. Appelbe and C. E. McDowell. Anomaly reporting – a tool for debugging and developing parallel numerical applications. In *Proc. First International Conference on Supercomputers*, FL, Dec. 1985.

[4] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Sublok. The ParaScope editor: An interactive parallel programming tool. In *Proc. Supercomputing '89*, pages 540–550, Reno, NV, Nov. 1989.

[5] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 1993. to appear.

[6] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[7] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4), Winter 1988.

[8] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 1991.

[9] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R^n}$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, Oct. 1986.

[10] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[11] A. Dinning and E. Schonberg. An evaluation of monitoring algorithms for access anomaly detection. Ultracomputer Note 163, Courant Institute, New York University, July 1989.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, Mar. 1990.

[13] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Apr. 1991.

[14] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[15] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proc. of Supercomputing '91*, pages 24–33, Albuquerque, NM, Nov. 1991.

[16] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Palo Alto, CA, Apr. 1991.

[17] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin — Madison, 1991.

[18] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *1992 Intl. Conference on Parallel Processing*, Aug. 1992.

[19] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, 1991. Also in *Proc. of the 3rd Workshop on Prog. Langs. and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

[20] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 133–144, Apr. 1991.

[21] I. Nudler and L. Rudolph. Tools for efficient development of efficient parallel programs. In *First Israeli Conference on Computer Systems Engineering*, 1986.

[22] E. Schonberg. On-the-fly detection of access anomalies. In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, June 1989.

[23] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proc. of the 1990 Symposium on the Principles of Programming Languages*, pages 218–231, Jan. 1990.

[24] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[25] V. J. Torczon. Multi-directional search: A direct search algorithm for parallel machines. Technical Report TR90-7, Department of Mathematical Sciences, Rice University, May 1989.

[26] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct. 1988.