Abstract. Source-level instrumentation allows a programmer to construct meaningful views of performance by associating measurements with specific program constructs. Often, however, these measurements are limited to the routine level because fine-grained instrumentation and compiler optimizations can interact to produce inaccurate results: optimizations can invalidate the results of instrumentation by inappropriately moving instructions in or out of the range of measurement and instrumentation can inhibit the application of optimizations, altering the executed code and distorting views of its performance. Existing tools avoid these issues by allowing instrumentation only at the routine level where restructuring optimizations are not applicable. We introduce the notion of “instrumentation-aware” compiling that allows accurate instrumentation down to the granularity of source-level statements and expressions despite aggressive optimizations. We discuss the implementation of a prototype instrumentation-aware, optimizing C compiler that preserves the placement of performance timer operations. We demonstrate its use, not only in C code, but also in the more complex, multilevel compilation environment for ZPL code.

1. Introduction

Source-level instrumentation allows programmers to associate performance measurements with specific program constructs. Often, however, these measurements are limited to the entry and exit points of routines, because to go below that level — to track the execution of individual expressions, statements, or basic blocks, for example — produces inaccurate results. This is because instrumentation and compiler optimizations can interact. On one hand, optimizations can invalidate the results of instrumentation by inappropriately moving instructions in or out of the range of measurement. On the other hand, instrumentation can inhibit the application of optimizations, altering the executed code and distorting views of its performance. In this paper, we introduce the notion of “instrumentation-aware” compiling which accurately places fine-grain, source-level instrumentation even in the presence of aggressive optimization.

An instrumentation-aware compiler preserves the semantics of instrumentation. For the purposes of this paper, we assume the semantics of the TAU profiling library[9][13][15] which contains standard object-oriented methods to register, start, and stop timers. Other types of instrumentation can be accommodated by appropriate changes in the instrumentor and compiler back end as discussed below.

In the next section, we further clarify the pitfalls of naive source-level instrumentation. In Section 3, we introduce instrumentation-aware compilation. In Section 4, we describe a prototype instrumentation-aware compiler, and in Section 5, we discuss its application to high level, parallel ZPL code [16].

2. Inaccuracy of Source-Level Instrumentation

When an optimizing compiler encounters instrumentation, it has only two options: it can assume that such calls do not interfere with application code and proceed with code restructuring or it can assume that such calls could interfere with application code and inhibit restructuring. In either case, the resulting measure-
ments may no longer reflect the behavior of the original program prior to instrumentation. We give two examples.

**Example: Instrumented C Code.** Figure 1 shows a C program that has been instrumented with TAU calls to time the single assignment statement with a timer named ft. The program is shown before compilation on the left, and after restructuring by a commercial optimizing C++ compiler on the right. The restructuring moves the invariant calculation of \((b\times c)\) out of the loop as indicated by the arrow. This transformation is safe — \(b\) and \(c\) are local variables that cannot be altered by the external TAU routines — but its results are misleading because part of the calculation specified by the programmer is not included in the measurement. On the other hand, if the initial C code is changed just slightly as in Figure 2, to involve global, rather than local variables, the code restructuring does not take place. The compiler recognizes that the TAU routines could potentially alter the global variables \(x\), \(y\), and \(z\), and therefore, does not move \((y\times z)\) out of the loop. The presence of instrumentation has inhibited optimization, altering the executed code that is to be measured.

These same interactions between instrumentation and optimization can be magnified in more complex, aggressively optimized code.

**Example: Instrumented ZPL Code.** ZPL is a high level, parallel array language that permits sophisticated optimization. A program is first optimized by a ZPL compiler during a source-to-source translation to C and then again by a C compiler during the translation into object code. For this example, we focus just on the ZPL to C translation, but the full compilation process is considered in Section 4 below.

---

1. This figure was generated by lowering the intermediate representation of the program after optimizations. The compiler generated symbols in the code were translated to a more readable, equivalent form by hand.
void tau_start_timer(void *);
void tau_stop_timer(void *);
void *ft;
extern int y,z;
int foo()
{
    int i, x;
    for (i=0; i < 500; i++)
    {
        tau_start_timer(ft);
        x = (y*z)+i;
        tau_stop_timer(ft);
    }
    return x;
}

extern void tau_start_timer(void);
extern void tau_stop_timer(void);
extern int y,z;
void *ft = 0;
extern int foo(void);
int foo(void) {
    auto int i; auto int x;
    i = 0;
    for(; i < 500; i+=1)
    {
        tau_start_timer(ft);
        x = ((y*z)+i);
        tau_stop_timer(ft);
    }
    return x;
}

Figure 2. Unoptimized code (left) and optimized code (right) demonstrates the inhibiting of the loop fusion optimizations.

Figure 3 shows a ZPL code segment that has been instrumented to measure the cost of individual array statements using three timers (ft1, ft2, and ft3). The upper case variables used in these statements represent conforming 2D arrays and, thus, the statements translate in a straightforward manner into doubly nested loops that iterate over the indices. If the code had not been instrumented, the ZPL compiler would have applied a loop fusion optimization, restructuring these statements into one loop nest. In the presence of the instrumentation, however, this restructuring becomes problematical. Figure 4 shows what happens if loop
fusion is applied to the instrumented code: the instrumentation is inadvertently corrupted so that all of the time is attributed to the first timer, \( ft1 \). Most compilers, including the ZPL compiler, however would not perform loop fusion under these circumstances. The ZPL compiler actually produces the code shown in Figure 5 where there are three separate sets of nested loops, one for each source statement. This version, while it measures the code the user specified, does so in a different program than the user was running before instrumentation! The new, unstructured program is significantly less efficient than it would have been because of the extra loop overhead and because fused loops expose further opportunities for optimization. The difference can be significant; in four processor run of this small program, the restructured code — much closer to the actual code the user was running before instrumentation — is 10% faster.\(^2\) This is a difference that could be critical in performance tuning.

The compiler writer is thus faced with a dilemma: suppressing an optimization may distort the underlying code significantly but proceeding with the optimization may not preserve the semantics of the instrumentation. Existing tools avoid these issues by allowing instrumentation only at the routine level where restructuring optimizations are not applicable. We propose a mechanism that would allow accurate instrumentation down to the granularity of source-level statements despite aggressive optimizations.

---

1. This code, as well as that of Figure 5 was produced by replacing names in the output of the ZPL compiler to increase readability.
2. The program was run on a network of four workstations. The mean wall-clock time for the unoptimized case was 109.8 seconds while that for the optimized case was 99.5 seconds.
3. Instrumentation-Aware Compilation

A typical compiler has three main components: a front end that parses the source code to build an internal representation, an optimizer that transforms that representation into a more efficient equivalent, and a back end that produces the target code. To build an instrumentation-aware compiler, each of these traditional components is modified and two new components — a de-instrumentor and an instrumentor — are added as shown in Figure 6. The input to the instrumentation-aware compiler is instrumented source code, and the output is target code with accurately placed instrumentation. The input is first fed into a de-instrumentor which strips instrumentation from the source code, collects symbol information for timers and their arguments, and extracts an instrumentation specification. The de-instrumented program is then fed into a standard compiler front end that produces an internal representation of the program, here called a code list, and an initial mapping table that relates elements of the code list to source code locations. The code list is then

```c
    do {
    TAU_PROFILE_START(ft1);
        {
    for (i0 = 0; i0 < N; i0++)
        {
        for (i1=0; i1 < N; i1++)
            {
            X[i0][i1] = A[i0][i1] + (B[i0][i1] - C[i0][i1]);
            }
        }
    TAU_PROFILE_STOP(ft1);
    TAU_PROFILE_START(ft2);
        {
    for (i0 = 0; i0 < N; i0++)
        {
        for (i1=0; i1 < N; i1++)
            {
            Y[i0][i1] = D[i0][i1] + (B[i0][i1] - C[i0][i1]);
            }
        }
    TAU_PROFILE_STOP(ft2);
    TAU_PROFILE_START(ft3);
        {
    for (i0 = 0; i0 < N; i0++)
        {
        for (i1=0; i1 < N; i1++)
            {
            Z[i0][i1] = X[i0][i1] + Y[i0][i1];
            }
        }
    TAU_PROFILE_STOP(ft3);
    iter = iter + 1;
    } while (! (iter > 100));
```

**Figure 5.** Code produced by the ZPL compiler that inhibits restructuring loop fusion in the presence of instrumentation.

---

**Figure 6.** Diagram of the instrumentation-aware compiler.
1:     #include <stdio.h>
2:     #include <Profile/Profiler.h>
3: 
4:     int x, y;
5:     int f(int a, int b, int c, int d)
6:     {
7:       int w, z;
8:       TAU_PROFILE_TIMER(ft, "Contribution of x,y and z", "[f()]", TAU_DEFAULT)
9:       TAU_PROFILE_START(ft);
10:      x = a + a * (b - c);
11:      y = (b - c) * d;
12:      z = x + y;
13:      TAU_PROFILE_STOP(ft);
14:      w = 5*z+6;
15: 
16:      return w;
17:   }

Figure 6. Structure of an instrumentation-aware compiler.

Figure 7. Using a timer to measure the cost of executing three statements.
passed to the optimizer which restructures it, updating the mapping table as necessary. At the end of this process, the mapping table relates each element of the code list to one or more source statements. The mapping tables, optimized code list, and the extracted instrumentation specification are then fed into an instrumentor which re-introduces the required instrumentation at the appropriate locations. The output of the instrumentor is sent to the compiler back end which generates target code.

Instrumentation-aware compilation avoids the problems affecting the accuracy of source-level instrumentation: instrumentation is not corrupted because it does not undergo restructuring, and optimizations applicable to the uninstrumented code are not inhibited because it is only the deinstrumented code that is optimized.

The maintenance of mapping information relating original source-level statements to the set of instructions in the optimized output is crucial. Consider, for example, the C program of Figure 7 which uses one timer to measure the cost of executing three assignments (to \( x \), \( y \), and \( z \)). Figure 8 shows Sparc assembly code for the routine \( f \) generated by a standard compiler without optimization. The routine \( f \) (starting at line 10), first
creates the timer \( ft \) (line 24), and then starts it (lines 25-28). After that, \( x, y, \) and \( z \) are calculated (lines 29-37, 38-44, and 45-49 respectively) and the timer is stopped (lines 50-53). Finally, \( w \) is computed (lines 54-56). The unoptimized assembly code thus measures exactly what the user would expect: the cost of the assignment to \( x, y, \) and \( z \) but not the assignment to \( w \). When the same program is optimized, however, the assembly code does not meet the user’s expectations as shown in Figure 9. There are two discrepancies. First, the compiler performs common subexpression elimination, computing \((b-c)\) (line 18) and using it (line 20) \textit{before the timer is created} (line 29) or \textit{started} (line 33). Second, the compiler performs the computation of \((z*5)\) (line 47) \textit{before timer is stopped} (line 51). Both optimizations are legal (because no application variables are modified in the timer operations) but they do not preserve the semantics of the source-level instrumentation: the cost of \((b-c)\) is not counted when it should be and the cost of \((z*5)\) is counted when it shouldn’t be.

Instrumentation-aware compilers can avoid these problems by using the mapping tables to accurately reintroduce instrumentation. Consider, for example, Figure 10 which shows a schematic of the mapping between the source code in Figure 7 and its object code in Figures 8 and 9. Blocks of object code are shown as rectangles shaded to indicate their mappings; unoptimized code is on the left and optimized code is on the right. In

\begin{verbatim}
1: .section "data"
2: .size 3,4
3: .align 4
4: .word 0x0
5: .global f
6: .section "text"
7: .align 4
8: .proc 4
9: f:
10: save %sp,-112,%sp
11: st %i0,[%fp+68]
12: st %i1,[%fp+72]
13: st %i2,[%fp+76]
14: st %i3,[%fp+80]
15: ld [ %fp+72],%i5
16: ld [ %fp+76],%i4
17: sub %i5,%i4,%l0
18: ld [ %fp+68],%i5
19: smul %l0,%i5,%l1
20: set .3,%o0
21: mov %o0,%o0
22: set .L4,%o1
23: mov %o1,%o1
24: set .L5,%o2
25: mov %o2,%o2
26: set 0xffffffff,%o3
27: mov %o3,%o3
28: call tau_profile_c_timer; nop
29: set .3,%i5
30: ld [%i5],%o0
31: mov %o0,%o0
32: call tau_start_timer; nop
33: set x,%i5
34: ld [%fp+68],%i4
35: add %i4,%i11,%i4
36: st %i4,[%i5]
37: set y,%i5
38: ld [%fp+80],%i4
39: smul %i4,%i10,%i4
40: st %i4,[%i5]
41: set x,%i5
42: ld [%i5],%i5
43: set y,%i4
44: ld [%fp+80],%i4
45: add %i5,%i4,%i10
46: smul %i10,5,%l0
47: set .3,%i5
48: ld [%i5],%o0
49: mov %o0,%o0
50: call tau_stop_timer; nop
51: .L2:
52: add %l0,6,%i0
53: ret; restore
54: .type f,#function
55: .size f,-f
\end{verbatim}

\textbf{Figure 9. Optimized Assembly Code for Routine f.}
the unoptimized case, the mapping is straightforward: a single block of contiguous code is mapped to each consecutive C statement. In the optimized case, however, the mapping is less straightforward: a block of code before the start timer (the calculation of \((b-c)\)) maps to both statement 11 and statement 12, and more than one block of code maps to each of statements 11, 12, and 15. (The mapping is thus many-to-many.) Using such a mapping, the instrumentor associates timers with each element of the code list and then the back end introduces code to turn the timers off and on for each segment as needed. The details of this are implementation-specific and we defer showing the resulting code until Figure 13 which comes after our discussion of our prototype implementation.

4. PROTOTYPE IMPLEMENTATION

Our prototype of an instrumentation-aware compiler supports C programs with TAU profiling instrumentation. We began with the source code of the compiler used in the FULLDOC project [4][7] which extends the publicly available and portable lcc C compiler [3] to include optimizations and support for debugging. It performs statement- and loop-level optimizations such as constant propagation, loop invariant code motion, dead code elimination, partial redundancy elimination, register allocation, and folding. In addition, it maintains mappings that relate source code locations to corresponding locations in the optimized code through multiple phases of code transformations [6]. Starting with this compiler, we describe the changes and additions needed to implement our instrumentation-aware compiler.

De-instrumentor. The de-instrumentor removes instrumentation, producing a clean version of the source code, a specification of the requested instrumentation, and symbol mappings. We implement it as a source-to-source translator that recognizes instrumentation by the TAU prefix. Timers may be created inside a routine (as they were in the code of Figure 7, for example) and follow normal scoping rules. The specification produced by our de-instrumentor, records both location and scope. A location is a tuple containing the source file name, the row (or y co-ordinate) and the column number (or x co-ordinate) in the source file. The properties of timers (the group to which they belong, name, symbol and their scope) are stored separately in an instrumentation symbols file that is compiled along with the de-instrumented source file. Entities in the symbols file are static variables with global scope. They are used as arguments for timer declarations and thus do not interfere with any optimizations in the compiler. The de-instrumentor replaces each instrumentation annotation with a blank line, retaining the original line numbering scheme of the annotated source code.
The de-instrumentor for our prototype makes two passes through the code. On the first pass, it uses the lcc compiler with code generation suppressed to get the scope of timers with respect to routine and symbol information. This enabled us to maintain consistency in routine naming conventions. On the second pass, it uses a simple token parser based on lex and yacc unix tools to separate instrumentation constructs from the source code, building the deinstrumented source, the instrumentation specification, and symbol mappings for the compile phase. Local names are prefixed with routine names to construct timer names with global scope; thus, for example, the timer \( ft \) in the routine \( f \) of the program in Figure 7, is called \( f_{-}ft \) in the global scope.

**Front End.** The compiler front end is modified to produce the initial mapping tables that relate code list elements to source locations. This information is already available in many compilers because it is needed by a variety of programming tools, in particular debuggers which use it to set breakpoints. In our case, the tables were, for the most part, already provided by the FULLDOC compiler. In some cases (ARG and CALL, for example), we had to make extensions to the instruction data structure for saving additional source information. We implemented a procedure to return the source code information for a given code list instruction. This provided a clean, uniform interface to source information that is otherwise embedded in the compiler in a variety of places: in debugging tables that contain def-points corresponding to source code locations, for example, or in tables that summarize def-use analysis, or in the instruction data structures themselves.

Elimination of Global Common Subexpression: duplicate expressions are eliminated so that the expression is computed once.

\[
\begin{align*}
\text{U1: } x &= a+b; & \text{O1: } u &= a+b; \\
\text{U2: } y &= a+b; & \text{optimized} & \text{O2: } x &= u; \\
& & \text{O3: } y &= u;
\end{align*}
\]

\( S(O_1) = \{U_1, U_2\} \)
\( S(O_2) = \{U_1\} \)
\( S(O_3) = \{U_2\} \)

Loop Invariant Code Motion: remove statements from within loops where computed values do not change within the loop.

\[
\begin{align*}
\text{U1: } \text{for (i=0; i<N; i++)} & \quad \text{O1: } x = c; \\
\text{U2: } \{ & \quad \text{O2: } \text{for (i=0; i<N; i++)} \\
\text{U3: } x &= c; & \text{optimized} & \text{O3: } \{ \\
\text{U4: } b[i] &= a + i + 3; & \text{O4: } b[i] &= a + i + 3; \\
\text{U5: } \} & \quad \text{O5: } \} \\
\end{align*}
\]

\( S(O_1) = \{U_3\} \)
\( S(O_2) = \{U_1\} \)
\( S(O_3) = \{U_4\} \)

**Figure 11. Source-level mappings between unoptimized code (left) and optimized code (right) represented as U and O labels respectively for two optimizations.**

\( S(O_i) \) is the set of source statements that generated \( O_i \).

**Optimizer.** The mapping tables must be maintained through optimizing transformations. Figure 11, for example, shows the mappings needed to track two common optimizations, common subexpression elimination and loop invariant code motion. Similar mappings can be constructed for each optimizing transforma-
tion in a straightforward manner. The FULLDOC compiler already maintained a lot of this information. To get the rest, we made several straightforward extensions to the restructuring routines of the compiler, generating more complete mappings.¹

Instrumentor. The instrumentor interprets start and stop annotations in the instrumentation specification to mean that all instructions with source correspondences in the delimited region of code are to be instrumented. This interpretation is tool-specific and may vary for different kinds of instrumentation. Our instrumentor modifies the code list so that each instruction has an ordered list of timers associated with it. The properties of each timer include its qualified name, operation (start or stop) and location (before or after the instruction executes). The algorithm we use is shown in Figure 12.

Instrument:

for each timer t
  for each source range r of t
    state = INACTIVE; previous = null;
    for each instruction i
      if i belongs to r then
        if state is INACTIVE then
          allocate instrumentation to start t before executing i
          previous = i; state = ACTIVE;
        end if
        if state is ACTIVE then
          previous = i;
        end if
      end if
    end for
  end for
end for

---

¹ In order to improve the coverage of the mappings, we also implemented some “clean up” heuristics but there are still some lapses in the implementation that result in unmapped instructions.

Back end. The back end converts the instructions to assembly code which is passed to the assembler. In our prototype, it is responsible for interpreting instrumentation annotations well. The back end examines the timers associated with the code list instructions and generates instrumentation using a fast breakpoint scheme [8]. To plant a fast breakpoint before an instruction that calls a measurement library routine, the scheme replaces the instruction with a branch to the breakpoint. At the breakpoint, state registers are saved and the measurement routine is called. After it executes, registers are restored, the replaced instruction is
executed, and normal execution resumes. We were able to exploit the fact that our measurement routines do not perform floating point operations and avoid the expense of saving and restoring them.

Figure 13 shows, for example, the final code produced by our prototype for the optimized code of Figure 9. Dashed arrows in the figure show the path of execution. After entering routine \( f \) (line 5), the control branches to a fast breakpoint (FB1_f at line 36) where the registers are saved (line 38) and the timer \( f_t \) (called \( f_f \) here as its scope is within routine \( f \)) is created (lines 40-50); control then returns to the application code (line 7). Before the computation of \( (b-c) \), control jumps again to a breakpoint, FB2_0 (line 32), after which point the timer \( f_t \) is started (line 37), \( (b-c) \) is computed, and control returns to the application routine (FB1_0 line 15). \( x \) is then computed (lines 17-22), multiplying \( (b-c) \) by \( a \) (line 20) and adding \( a \) (line 21). Next, \( y \) is computed (lines 28-29), re-using \( (b-c) \) and multiplying it by \( d \) (line 25). \( z \) is computed by loading \( x \) and \( y \) (lines 27-30) and then branching to the fast breakpoint FB2_3 (line 61) where \( z \) is computed as \( (x+y) \) (line 62) before stopping the timer \( f_t \) (line 67) and returning to the application routine (FB1_3 line 32). This is followed by computing \( w \) (lines 33-34), multiplying \( z \) with 5 (line 33) and adding 6 to the result (line 34). Thus, the timing of \( f_t \) includes the complete computation of \( x \), \( y \), and \( z \) and excludes the computation of \( w \) as the instrumentation specifies. This assembly file is then converted to binary code by the assembler. When the binary file is linked with measurement libraries and executed, it generates

\[
\begin{align*}
\text{start()}; & \\
\text{branch FB1_0}; & \\
x &= a + a \cdot (b-c) & (b-c) \text{ branch FB1_3}; \\
y &= (b-c) \cdot d & (b-c) \text{ branch FB1_3}; \\
z &= x + y & (b-c) \text{ branch FB1_3}; \\
w &= 5 \cdot z + 6 & (b-c) \text{ branch FB1_3}; \\
\end{align*}
\]
the performance data in Figure 14 showing the time measured by the timer \( ft \) ("Contribution of \( x, y \) and \( z[f()] \)).

This scheme is light weight as compared to unix level breakpoints that involve the use of a signal (SIGTRAP) and have an overhead of the order of milliseconds. Using our prototype, the overhead of a pair of TAU start and stop calls is just 2.16 microseconds. (This cost was determined by amortizing over ten million calls to a profiled routine and includes both the TAU overhead for profiling as well as the fast breakpoint overhead as observed on a Sun UltraSPARC-IIIi processor running at 440 MHz under the Solaris 8 operating system.)

5. Applications of the Instrumentation-Aware Compiler to High-Level Code

Programs written in high level, parallel languages such as ZPL or HPF are typically compiled in multiple stages using successive optimizers and then linked with runtime system libraries to produce executable images. A program written in ZPL, for example, is first optimized by the ZPL compiler as it is translated into C code, and then by the C compiler as it is translated into object code. After compilation it is linked to a complex runtime system that uses both a portable layer, IronMan [2] and the MPI library [11]. In this section we demonstrate that instrumentation-aware compilation can be effective in providing support for multi-level profiling even in this complex compilation environment.

To fully support instrumentation at the ZPL source level, we will need to compose instrumentation-aware compilers for both ZPL and C as shown in Figure 15. For the purposes of this example, we did not implement a full version of an instrumentation-aware ZPL compiler, but instead, made a simplifying assumption, used mapping information generated for a previous debugger, and augmented the result by hand where necessary. Our simplifying assumption was that all program statements were to be individually profiled which allowed us to skip the deinstrumentation phase. We used mapping tables produced for the ZEE debugger.
**Figure 15. Multistage Instrumentation-Aware Compilation in ZPL.**
and then “reinstrumented” the C code manually. The effect of this was that we were able to translate a ZPL program into the instrumented C program that would have been produced by an instrumentation-aware ZPL compiler. For the Jacobi ZPL program in Figure 16, for example, we produced the C program in Figure 17, which was then compiled by our instrumentation-aware C compiler.

This two step compilation provided us with good coverage for the source code, but to understand the runtime behavior of parallel code, it is often necessary to consider communication costs. The ZPL runtime system uses the IronMan communication layer as well as the MPI library. To measure the contribution of IronMan, we again used our instrumentation-aware C compiler, this time to instrument the IronMan routines. To target the contributions from MPI, we did not have access to source code and so used the MPI Profiling interface which generated a wrapper interposition library for its instrumentation.

This multi-level instrumentation targeted a common measurement API, allowing streams of performance data to flow into a common performance data repository to present a seamlessly integrated picture. The interfaces cooperate with each other: the MPI wrapper library, for example, provides information on the rank of the MPI processes, and the compiler based instrumentation interface uses this information to store data in appropriate files after the top level routine exits from the program. Figure 18 shows a profile of the optimized ZPL program. Note that ZPL program statements, such as “[R] A := 0;” and “err := max<<fabs(A-Temp)” are integrated in the profile with the ZPL runtime system entities, such as “_SetupRegionWithDirections,” and MPI level entities, such as “MPI_Recv().” Instrumentation is inserted in the program at the source-level, the compiler level, the MPI wrapper library level and the runtime system library level. A multi-level instrumentation scheme provides better flexibility in composing a performance experiment and covers observable behavior more completely [12][13][14]. This ZPL case study effectively uses multi-level instrumentation for wider coverage. It demonstrates the viability of applying our instrumentation-aware approach to multiple stages of optimizations to form an optimized executable that includes both fine-grained statement-level and coarse-grained routine-level instrumentation.

```plaintext
17 procedure Jacobi();
18 begin
19  [R] A := 0.0;     -- Initialization
20  [north of R] A := 0.0;
21  [east of R] A := 0.0;
22  [west of R] A := 0.0;
23  [south of R] A := 1.0;
24  [R] repeat      -- Body
25    Temp := (A@north+A@east+A@west+A@south)/4.0;
26    err := max<<fabs(A-Temp);
27    A := Temp;
28    until err < delta;
29
30  [R] writeln(A);
31 end;
```

**Figure 16. Source code of ZPL Jacobi program.**
6. CONCLUSIONS

Source-level instrumentation allows a user to explore a program’s performance in terms of familiar and meaningful language-level constructs. Often, however, this exploration is unnecessarily constrained because environments only support instrumentation at routine boundaries. To go below that level — to track the execution of individual expressions, statements, or basic blocks, for example — introduces inaccuracies due to interactions between compiler optimizations and the instrumentation. We have introduced an extension to traditional compilers which avoids these problems, preserving the semantics of fine-grained instrumentation despite the presence of aggressive optimization. We have demonstrated the effectiveness of the approach with a prototype C compiler and with ZPL, a modern high-level, parallel programming environment with sophisticated optimizations and an extensive runtime system.

Instrumentation-aware compilation gives the user flexibility to fully explore the behavior of a program but it is not a panacea. There are two potential problems with the accuracy of instrumentation that we can not address. The first is perturbation. All instrumentation introduces overhead and, in the case of fine-grained instrumentation, this overhead can be relatively high in comparison to the cost of the code being measured. In parallel environments, the resulting perturbation can be significant. Mechanisms for removing perturbation from measurements exist and may be productively applied both at the level of performance analysis and at the level of instrumentation-aware compilation: the compiler could, for example, warn the user of situations where the ratio of perturbation to measured code rises above some threshold. These adjustments are beyond the scope of this work. The second potential problem is that instrumentation can be cor-

```c
#include "jacobi_usr.h"

void Jacobi() {

    TAU_PROFILE_TIMER(t19, "[R] A := 0.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t20, "[north of R] A := 0.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t21, "[east of R] A := 0.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t22, "[west of R] A := 0.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t23, "[south of R] A := 1.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t25, "[R] repeat", "", TAU_USER);
    TAU_PROFILE_TIMER(t26, "Temp := (A@north+A@east+A@west+A@south)/4.0;", "", TAU_USER);
    TAU_PROFILE_TIMER(t27, "err := max<< fabs(A-Temp);", "", TAU_USER);
    TAU_PROFILE_TIMER(t28, "A := Temp;", "", TAU_USER);
    TAU_PROFILE_TIMER(t31, "[R] writeln(A);", "", TAU_USER);

    /* begin MLOOP *** line 21 of jacobi.z */
    for (_i0 = _REG_MYLO(_R_east_of_R,0); _i0 <= _REG_MYHI(_R_east_of_R,0); _i0 +=
        _REG_STRIDE(_R_east_of_R,0)) {
        _i1 = _REG_MYLO(_R_east_of_R,1); _i1 <= _REG_MYHI(_R_east_of_R,1); _i1 +=
            _REG_STRIDE(_R_east_of_R,1)) {
            TAU_PROFILE_START(t21);
            (*((double *)_F_ACCESS_2D(A,_i0,_i1))) = 0.0;
            TAU_PROFILE_STOP(t21);
        }
    } /* end MLOOP */

    FIGURE 17. INSTRUMENTED C CODE FOR THE ZPL JACOBI PROGRAM.
```
ruptured after compilation, by peephole optimizations in assemblers, for example, or the out-of-order execution of instructions due to modern, sophisticated pipelines and caches. It may be possible to address assembly level issues with an instrumentation-aware assembler. It will not be possible to address all hardware optimization issues with this approach. Thus, while we have removed much of the inaccuracy introduced by source-level instrumentation, some will remain, affecting fine-grain more than course-grain requests. We have given users the flexibility of determining the granularity of their instrumentation but, because of the potential for both perturbation and execution order inaccuracies, they will have to be aware of the pitfalls of relying on extremely fine-grained measurements.

Finally, although we have demonstrated the use of instrumentation-aware compilation with a profiling package, but other types of instrumentation can be easily supported with appropriate changes to the instrumentor and back end.

Acknowledgments. We would like to thank Bradford Chamberlain, Clara Jaramillo, Lawrence Snyder, Mary Lou Sofka, and Allen Malony for their many useful insights during discussions of this material. We would also like Clara Jaramillo and Mary Lou Sofka for the use of the FULLDOC compiler, and for their assistance in helping us understand its inner workings.
REFERENCES