Java and Scientific Programming: Portable Browsers for Performance Programming

Michał Cierniak
cierniak@cs.rochester.edu
Department of Computer Science
Univesity of Rochester
Rochester, NY 14627

Suresh Srinivas
ssuresh@engr.sgi.com
Silicon Graphics Incorportated
2011 North Shoreline Blvd
Mountain View, CA 94040

Abstract

We present *jCITE*, a performance tuning tool for scientific applications. By combining the static information produced by the compiler with the profile data from real program execution, jCITE can be used to quickly understand the performance bottlenecks. The compiler information allows great understanding of what optimizations have been performed. The user can also find out which optimization have not been applied and why.

Platform independence makes Java the ideal implementation platform for our tool. SGI users can have the same performance analysis tool on all platforms. You can run jCITE on an SGI O2 computer to optimize code for that machine, or you can run jCITE from within Netscape on a SPARCstation to analyze performance of a Cray application.

In our experiments we were able to significantly speed up some SPEC95 applications in a few days or even a few hours without any prior knowledge of those applications.

1 Introduction – the need for performance programming for Scientific Computing

Performance is at the heart of scientific and engineering computing. Tuning a critical inner loop could lead to drastic reduction in the running time of a large scientific or engineering application. This paper is about portable performance programming tools for the scientific and engineering programmer.

Larry Carter and Bowen Alpern define *performance programming* as "programming when a 30% improvement in speed may be worth weeks of work" [1]. According to their work obtaining maximum application performance on a given platform is difficult. Completely automatic optimizations, while very effective, do not fully solve the problem. This is evident in the complex optimization options used to compile standard benchmarks, like SPEC95 [10]. By performance programming we understand tuning applications with very demanding performance requirements. Performance programming usually requires manual tuning of the source code and compiler options to obtain the fastest code. Having a number of performance programmers is common in hardware companies whose focus is scientific and engineering computation. They help tune a number of large ISV (Independent Software Vendor) applications such as LS-DYNA3D—a large Fortran application which is used in a variety of applications from automobile design and safety to biomechanics.

The unfortunate reality is that performance programming is hard. The advent of optimizing compilers which perform various transformations on the program source to utilize the memory hierarchy better has made performance programming even harder. A performance programmer usually has the following in their arsenal:

- Algorithmic and computational knowledge in specific areas such as Fluid dynamics, Operations Research, Structural Mechanics, Computational Chemistry.
- Good understanding of various compiler options (flags), computer architecture, and assembly programming.
- Dynamic tools such as prof, pixie, and hardware counters (in newer microprocessors such as the MIPS R10000, the Digital Alpha 21164, and the Sun UltraSPARC)—to understand the profile of their application and identify subroutines or functions to focus their attention.

We have designed jCITE—A Java based browser—to alleviate the performance programming problem. jCITE helps optimize applications whose performance are difficult to understand. It does so by adding several things to the performance programmers arsenal:

- Details of compiler optimization or optimization failure (non-optimizations) information at the level of the application source.
- Visual metaphors for correlating original source, compiler transformed source, and runtime performance information.
- Ability to query for detailed compiler optimization information at application hot-spots (elsewhere as well).
- Provides a portable browser to be used in a heterogeneous environment (e.g., connecting to an SGI Origin2000 through an SGI O2, Apple Power Mac, or a Compaq Presario).

jCITE uses profiling information to find program fragments which the performance programmer has to to concentrate on. If the hardware allows it, the same runtime information can be also used to make a rough judgment on the nature of the performance problem (R10000 performance counters [11, 9] can measure and classify cache misses, floating point operation and other processor events useful for understanding the performance). The runtime information is combined with the information produced by the compiler about applied optimizations/non optimizations. jCITE **does not**

- Assist the performance programmer in choosing data structures or algorithms.
- Allow editing of the program. We assume that available editors are used for that purpose.

In the jCITE browser we combine both the compile time and runtime performance information in a visually appealing way. We introduce a new visual metaphor known as *Synchronization*. Say you have two text windows one containing the original program source and the other containing the compiler transformed source. By *Synchronization* them we mean that when the mouse is moved over a region of text in the original source window, regions of text that are compiler transformed (derived from that region) are highlighted on the second window and vice versa. We also have borrowed other metaphors such as hypertext and context sensitive pop-up menus that have become popular in Web browsers.

The paper is structured into the following sections:

Case Studies:	where we illustrates how jCITE works using two examples familiar to scientific programming.
Design and Architecture:	where we describe the inner workings of the jCITE browser.
Experiments:	where we present results from using jCITE to do performance tuning of the compiler and a SPEC95 integer application.
Lessons Learned and Conclusions:	where we describe what we learned in the process as well as describe related and future work.

2 Case Studies

We walk through two examples (matrix multiply and SPEC95 multi grid) to illustrate some of the capabilities of the jCITE browser.

2.1 Example 1—matrix multiply

Multiplying two matrices is a recurring theme in scientific computing. We use a simple program that does matrix multiply in several different ways as well as multiply matrices of several different sizes to illustrate:

- Synchronizing original source and compiler transformed source.
- Querying static compiler information produced for loop nests and inner loops.

In **Figure 1** we can see a snapshot of jCITE when the matrix multiply fortran program is loaded in. jCITE automatically loads in any compiler transformed sources if the programmer had asked them to be produced by the compiler (the option `-LIST:cite` causes the SGI MIPS Pro 7.x compilers to produce the relevant files). In the snapshot the current line was 75 in the subroutine `mmjki200`. By default jCITE assumes that the programmer wants synchronization between the source and the transformed source. This can be turned off. In the same figure we see a number of lines on the right window highlighted in blue. These are the source lines that are in the compiler transformed source that came originally from line 75.

As we can see from the snapshot the compiler has performed a number of transformations on this small—8 line—subroutine. It has:

- Peeled out the initialization of the result matrix.
- Changed the order of the loops. "`i`" is now the inner loop.
- It has tiled the loops to utilize the memory hierarchy better. It has tiled for both first and second levels of caches.
- It has *register blocked* or *unroll and jammed* the inner tiles to use registers better as well as to make it better schedulable for the code generator

The compiler—in this case the loop nest optimizer—produces this information for jCITE in a separate file. **Figure 2** shows an example of what is produced for the loop nest on line 71, 72 and 74. Lisp-like format is used for historical reasons (jCITE had its origins in CITE which was written to work within Lucid's XEmacs).

2.2 Example 2—SPEC95 107.mgrid

The 107.mgrid program is part of the SPEC CPU95 (floating point) benchmark. It is a multi-grid solver in a 3D potential field. Its reference dataset is quite large but the program itself is just 500 lines of fortran77 code. It is also one of the easiest to parallelize by an automatic parallelizer.

We use it as a case study to illustrate:

- Annotating runtime performance information on the application source while maintaining correlation between application source and compiler transformed source.
- Automatic Parallelization Information. This also includes array region information when the parallelizer cannot parallelize the loop.
- Combining runtime performance information with compiler information.
- Annotating multiple runtime performance information on the application source.

Figure 3 shows a snapshot of jCITE when the mgrid program is loaded and the programmer asks to see the number of cycles (the histogram labeled `cy_hwc`) each line of mgrid has consumed. The cycles are measured using program counter sampling whenever the R10000 processor's internal cycle measurement counter overflows. The programmer can also ask jCITE to hypertext/annotate all the loops that were parallelized or were not parallelized. Notice how the scrollbars are also painted to show the program hot-spots.

For the loop nest we are seeing in the figure, the compiler has parallelized the outermost loop and so has partitioned the `I2` loop amongst all the processors. The figure also shows that the compiler has removed a number of array references out of the loop. This snapshot is from a pre-release version of the automatic parallelizer.

Figure 4 shows a snapshot of jCITE with mgrid but at a later point than **Figure 3**. The programmer has asked to see the cycle counts (`cy_hwc`), secondary cache misses (`dsc_hwc`) as well as compiler information on one of the loops. Notice how a new embedded graphics window shows the cache miss histogram and a new split window at the bottom shows the compiler information.

3 Design and Architecture of jCITE

Compilers alone cannot provide a fully automatic solution to the performance programming problem for the following reasons:

- Programmers often have high-level knowledge about the problem and the application domain which can be used for optimizations, but cannot be inferred from the source code alone.
- Some optimization techniques while possible in practice are not implemented in the compiler (or are implemented incorrectly).

The problem with manual tuning is that complex, scientific programs are difficult to understand—especially if the programmer who performs the tuning for a given platform is not the original author of the application, or if the application had multiple authors.

3.1 Design Goals

We address the difficulties encountered by application programmers by building a browser which will help understand the performance impact of compiler optimizations. The design of jCITE focused on the following goals:

- Integration of static information generated by the compiler with the dynamic information obtained by profiling the program.
- Presenting information at high-level as much as possible. Both the profiling and compiler information are mapped to the original source program, but it is possible to simultaneously see the same code fragment as transformed source or assembly, if the low-level information is necessary.
- Portability and smooth integration with Web Browsers. We wanted to be able to run jCITE on any platform. We also wanted to be able to make jCITE work under any Web Browser.
- Ease of use. Generation and use of the extra information by the compiler and the profiler should be as simple as possible.
- Extendibility. It should be easy to add new functionality if required by a given set of applications.

The rest of this section will show how we have achieved those goals from the point of view of specific design decisions.

3.2 Working with the jCITE browser

jCITE centers around the application source code. However, additional information is required to show anything more than the source code (jCITE itself does not perform any analysis of the source code—it combines information from the MIPS Pro compiler and the Speedshop run-time performance collection tools).

The work with jCITE usually follows the pattern of:

1. Compile the application with `-LIST:cite` option.
2. Run the desired experiments with Speedshop.
3. Analyze the program with jCITE. If satisfied, stop.
4. Modify the source, and or change compiler options.
5. Go to step 1.

MIPS Pro compilers save the information in files whose names can be derived from the source file names. Running Speedshop is optional but we highly recommend it since the run-time information often offers invaluable insight about the program behavior. We assume that the profiling information produced by Speedshop is saved in files whose names are derived from the application name and the monitored event name [9] (examples of *events* are: clock cycles, primary data

cache misses, secondary instruction cache misses, or floating point operations). It is the user's responsibility to save the profiling information in files with appropriate files names. This effect can be trivially achieved with a simple script since the file names we use follow the same conventions for event names as the Speedshop tools.

Upon start jCITE searches the appropriate directories for compile-time and profile information and automatically configures itself to use all available information.

3.3 jCITE: Why implement in Java?

Why did we choose to implement jCITE in Java?

1. Portability. We wanted the jCITE browser to be able to work on a variety of platforms. For example we wanted to be eventually able to connect to Origin2000 from SGI O2, Apple Power Mac or Compaq Presario using the jCITE browser.

2. High level component (Java Bean) availability for spreadsheets, and variety of graphics components for visual display of information. We wanted the jCITE browser to be extended by others to use components written by other ISV's developing for Java.

3. Live documentation accessible through Web Browsers. We wanted jCITE to be able to run within a Web browser. This would allow potential customers to get a feel for what the SGI Compilers and performance tools can do through the Internet.

Implementing in Java was fun but we encountered a number of performance problems which we will discuss in the final section.

3.4 Generating compiler information

The SGI MIPS Pro compiler has special options that allow it to produce listing files that describes the loop optimizations that it performed. Since the loop nest optimizations are fairly high level it maybe desirable to view the code after the loop nest optimizations. To accomplish this the SGI MIPS Pro compiler provides translators from its internal format back to Fortran or C. These translators can be invoked using listing options in the compiler. Other optimizations such as inner loop unrolling, software pipelining, and local scheduling produce compiler information by embedding them in the assembly file as comments.

The compiler produces a variety of information for jCITE. These include:

- **Loop Nest Information** including register, cache blocking, dependence problems, fission, and fusion.

- **Prefetch Information** which includes the array references that are prefetched and whether they are for the L1 or L2 cache, the confidence numbers for them and other details about them such as the volume that is prefetched in a given loop per iteration of the loop.

- **Structure of Loop Nests** which is a sketch of the loop structure after all the loop level transformations. This information helps identify whether a loop was a wind-down, regular loop or peeled loop. It also gives number of iterations the loop is going to execute or an estimate for the trip count.

- **Software Pipelining Information** which provides information about the number of cycles it takes to execute an inner loop. It also gives the number of integer operations, the floating point operations, as well as what percent of the processor peak the operations are executing. If the inner loop did not pipeline then reasons for that are also given. Similar information is provided by the scheduler for loops that are not inner loops.

- **Inner Loop Unrolling Information** gives the unrolling factor by which inner loops were unrolled. If they were not unrolled it lists why the inner loop was not unrolled.

- **Parallelization Information** tells what loops the automatic parallelizer expects to go in parallel as well as regions of various arrays that are written and read from.

3.5 Running and examining performance experiments

The MIPS Pro Compilers include the *Speedshop* package. Speedshop is the generic name for an integrated package of performance tools to run performance experiments on executables, and to examine the results of those experiments.

Experiments are recorded using the `ssrun` command, as follows:

```
ssrun -<exptype> <a.out-name> <a.out arguments>
```

where `<exptype>` is one of the named experiments. A common experiment is PC sampling. The program counter is statistical sampled, using 16-bit bins, based on user and system time, with a sample interval of 10 milliseconds. The size of the sampling interval and the size of bins can both be controlled. On the R10000 which has hardware performance counters, a variety of other statistical PC sampling experiments can be performed [11, 9]. These include (among others):

- **Cycle Counts** which uses statistical PC sampling, based on overflows of the cycle hardware performance counter.
- **Primary Data Cache Misses** uses statistical PC sampling, based on overflows of the primary data-cache miss counter.
- **Secondary Data Cache Misses** uses statistical PC sampling, based on overflows of the secondary data-cache miss counter.

The `prof` command is used to generate a report about a specified experiment. jCITE assumed that the prof command has been run to produce the performance experiment data file.

3.6 Inner workings of jCITE

The high-level structure of jCITE is presented in **Figure 6**. Upon start jCITE looks for all files which are associated with the application. First the source and transformed source files are read and the data structures with their representation are created. The a file which describes mapping between lines in the two sources is parsed with an S-Expression reader. The sources are annotated with appropriate mappings. The S-Expression reader is invoked again to read LNO (Loop Nest Optimizer) information. That information is attached to the internal representation of the source. Next the profile information produced by Speedshop is read and attached to corresponding sources.

The data structures are organized so that all this information can be quickly accessed. Whenever needed additional data structures are created to contain local information extracted from the global data structures. For example on a mouse click for a given source line, the data structures associated with that source file as well data structures for that specific line are searched and the information is merged into the pop-up menu data structure which contains labels to be placed in the menu and actions to be performed if a menu item is selected. This data structure is passed to the method which displays and handles pop-up menus.

Similarly, when the user adds a new histogram to the currently displayed source, a data structure for that histogram is created and inserted into the UI container displaying that source.

4 Experiments

In this section we present some of the experiments that we did using jCITE. We have not yet released jCITE to a wide audience.

4.1 Performance tuning the compiler

We present two examples. The first one is a real world application from the supercomputing group within SGI. The core of the application was a subroutine `mai_ope_Track3d`. They were not happy with the performance of this subroutine and wanted us to examine it closely. It had a inner loop that was 448 lines long. The MIPS Pro 7.x compiler fissioned this loop into 15 loops but the software pipeliner was able to pipeline only some of the loops. The assembly code was almost impossible to decipher since the Code Generator had unrolled all the loops and also had introduced unroll remainder loops. We extended jCITE with a simple script that told us the cycles each of the loops produced after the fissioning and whether

it was software pipelined or not. Using this information we were able to tune the loop fissioner to produce a better balance of fissioned loops which resulted in all the fissioned loops getting software pipelined.

We encountered the second example when we were tracking a regression in the SPEC95 applu benchmark between compiler releases. We used a combination of runtime performance information and compiler information to track and fix this problem. In one of the frequently executed subroutine, two subroutines it called were inlined together to produce a bigger routine that has two outer loops that iterate over the same space. The earlier version of the compiler was able to fuse the two loops together but the later version of the compiler was not. We identified the problem using jCITE as one that invoked the optimizer, a label that was introduced between the two loops due to inlining was not being removed causing the loop nest optimizer to not fuse the loops.

4.2 Hand tuning the SPEC95 130.li integer benchmark

We wanted to see if we could use jCITE to tune programs other than scientific programs. We took upon ourselves a challenge to see if we could improve an arbitrary integer program. We decided to choose the lisp interpreter 130.li for several reasons:

- We gave ourselves 3–4 days to tune the program and wanted a large enough program but not too large. 130.li was among the larger SPEC95 benchmarks.
- We wanted to give feedback to the optimizer group if we found anything interesting.
- SPEC95 programs are optimized only using compiler flags and we were curious to see how much better hand optimization could do.

We first profiled the program and on looking at the application hot-spots using jCITE we determined the following optimization opportunities:

Opt1 The compiler was not recovering common subexpressions across basic blocks. We recoded the macros `consp`, `listp` to use fast versions when their argument was evaluated to be not null in a prior basic block.

Opt2 Varargs routines were not being inlined. We specialized the varargs routine into 7 different versions and changed all the call sites to use the right version. We also cleaned up the body of the varargs routine to no longer loop through the varargs. For example `void func_vararg(int *args,...)` when used with one and two arguments could be recoded as `void func_1(int *arg1)`, and `void func_2(int *arg1, int *arg2)` respectively.

Opt3 Switch statements were compiled in a table driven fashion. We wanted to use 'if' statements in a few cases.

Opt4 If recursive procedures have fast return path for empty arguments the register allocator today spills registers on entry to function and restores them across the fast return path. We recoded the procedure to eliminate the condition check and moved the condition to the caller.

Opt5 We observed that some of the compares could be converted to bit operations to do the compares faster. This was an optimization that we do not believe a compiler would be able to do.

Opt2 and Opt4 needs an interprocedural optimizer. Opt5 is not possible through a compiler. Opt1 and Opt3 are doable by a good optimizer. Here are the experimental measurements on an Origin2000 machine with an R10000 processor. We used the hardware cycle counter to get an estimate of where to start looking.

Optimization attempted	Measured Time	Percentage improvement over base case (cumulative)
Original	119.616u 0.095s 2:00.30	0%
Opt1 (redundant checks)	119.199u 0.161s 1:59.98	*negligible*
Opt2 (no var args just in `xleval.c`)	113.797u 0.091s 1:54.48	4.8%
Opt2.1 (no var args everywhere)	107.821u 0.152s 1:48.57	9.8%
Opt3 (converted switch to if condition for `livecar/livecdr`)	101.505u 0.074s 1:42.08	15.1%
Opt4 (moved `if (arg == NULL) return;` out of `mark` to caller)	97.188u 0.079s 1:37.77	18.7%
Opt5 (bit operations instead of compares)	92.498u 0.077s 1:33.06	23%

As the table shows with 3–4 days of work we were able to get close to 23% improvement over the base case. We compiled the base case to use the new ABI, interprocedural inlining, `-O3` optimization, targeting the R10000 processor (`-Ofast`).

5 Lessons learned and conclusions

In this paper we have described a Java based browser that will be valuable for a performance oriented programmer. We are not aware of any software that provide such extensive compiler optimization information as well as combines runtime performance information with compiler optimization information in a visually appealing way either in academia or industry.

We learned several things from designing and implementing jCITE.

- Combining compiler information with runtime performance information can lead to new insights into how program performance matches the hardware.
- Novel visual display are much more attractive in presenting performance data than static text data. Correlating data to provide information is even more useful. Browsers for performance programming should strive for high bandwidth (more information) to the user.
- Java specific lessons:
  - Java based technology promises portability, better client software on Unix machines, and much more of a chance for ISV's to participate in providing tools for high end programming like scientific and engineering programming.
  - GUI portability has a price: performance. The state of art of Java compilers and components are not yet sufficient to develop large scale portable GUI software.
  - Using multi threading safe data structures such as Vectors [2] are expensive today on most platforms even when the application is single threaded. jCITE uses Vectors in a number of places.
  - The I/O performance of Java VM's we have used have mostly been insufficient for programs such as jCITE that read lots of files.
  - The lack of features in AWT (JDK 1.0.2) such as pop-up menus, cut and paste was disheartening. Several of these deficiencies have been fixed in JDK 1.1 [6]. Sun has also provided a road map for lightweight and fast UI components. The competition in the UI between Microsoft's AFC and Sun's JFC is going to lead to much better UI software similar to what we witnessed in Browser wars and the technology that was swept with it.
- It is better to design a lightweight browser from the ground up for performance programming than integrate functionality into a fully functional editor. jCITE's precursor CITE was written entirely in Emacs Lisp and worked within Lucid's XEmacs. CITE was heavyweight due to it's dependence on XEmacs. Also religious wars over choice of editors are best left to newsgroups than products.

5.1 Related Work

Sigma [8] was a project at Indiana University. It had an Emacs based front-end which allowed interactive transformations to be applied to a program. It had a library to examine and manipulate Intermediate Representation of the program. It was intended more as a restructuring tool than for performance programming.

The program analysis group at Microsoft Research [5] are developing advanced programming tools. Their programming tool is currently Emacs-based and uses the result of their intra- and interprocedural program analysis. Their system differs from ours in that their focus is on issues such as alias and dataflow analysis in the context of C and C++ programs. We also plan to incorporate results of the alias analysis from our compiler into our programming tool.

The Parascope programming environment, the D editor [3] and the subsequent work at Rice on compilation and performance evaluation environment is similar in spirit to ours. They focus more on data parallel programs in a message passing environment. We plan to incorporate results from data distributions for parallel programs running on Scalable Shared Memory machines that are being developed in our compiler into our programming tool.

SGI's Workshop MPF tool [7] provides information on KAI parallelizer optimizations. It is written to work only on SGI's. KAI [4] has announced a new suite of tools written in Java to provide portability. They are also working towards providing portability of the parallel directives across a number of platforms so that ISV's can migrate their parallel code easily across platforms.

References

[1] B. Alpern and L. Carter. Performance Programming: A Science Waiting to Happen. In U Viskin, editor, *Developing a Computer Science Agenda for High-Performance Computing*. ACM, ACM Press, 1994. Available from UCSD at `http://www-cse.ucsd.edu/users/carter/perfprog.html`.

[2] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.

[3] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and ScottWarren. The D Editor: A New Interactive Parallel Programming Tool. In *Proceedings of Supercomputing 1994*. IEEE Computer Society, November 1994. Available from the CRPC technical archive `http://www.crpc.rice.edu/CRPC/softlib/TRs_online.html`.

[4] Kuck and Associates. KAP/Pro Toolset. Availabe at `http://www.kai.com/kpts/`.

[5] Microsoft Program Analyst. The Microsoft Program Analysis Web Page. Available at `http://www.research.microsoft.com/research/analysts`.

[6] Sun Microsystems. The Java(tm) Developer's Kit, 1.0.2 and 1.1 versions, 1996, 1997.

[7] Silicon Graphics Technical Publications. Workshop MPF User's Guide. Available at `http://www.sgi.com/techpubs/`.

[8] B. Shei and D. Gannon. Sigmacs: A Programmable Program Restructuring Tool. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Workshop on Compilers for Parallel Systems, Aug. 1990, Irvine, CA*, pages 88–108. Pitman Publishing, 1991.

[9] Silicon Graphics, Inc. MIPS R10000 Performance Counters. Available at `http://www.sgi.com/MIPS/products/r10k/Perf_Cnt/R10K_PF_Count.doc.html`, March 1997.

[10] Standard Performance Evaluation Corporation. SPEC CPU95 Results. Available at `http://open.specbench.org/osg/cpu95/results`, March 1997.

[11] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing 1996*. IEEE Computer Society, November 1996. Available at `http://www.sgi.com/MIPS/products/r10k/App_notes/Perf_Count/index.html`.

mmj.f

```
58  C     call stop_timer(3)
59  C     call print_timer(3,'mmjik 1000')
60
61        print *,'c1(1000,1000)='
62        print *, c1(1000,1000)
63
64      stop
65      end
66
67      subroutine mmjik200(a,b,c)
68      parameter (n=200)
69      real*8 c(n,n),a(n,n),b(n,n)
70
71      do j = 1, n
72         do i = 1,n
73          c(i, j) = 0.0d0
74          do k = 1,n
75            c(i, j) = c(i, j) + a(i, k) * b(k, j)
76          enddo
77        enddo
78      enddo
79
80      return
81      end
82
83      subroutine mmjki200(a,b,c)
84      parameter (n=200)
85      real*8 c(n,n),a(n,n),b(n,n)
86
87       do j = 1, n
88         do i = 1, n
89           c(i, j) = 0.0d0
90         enddo
91         do k = 1, n
92           do i = 1, n
93             c(i, j) = c(i, j) + a(i, k) * b(k, j)
94           enddo
95         enddo
96       enddo
97
98       return
99       end
100
101     subroutine mmjik1000(a,b,c)
102     parameter (n=1000)
103     real*8 c(n,n),a(n,n),b(n,n)
104
105     do j = 1, n
106        do i = 1,n
107         c(i, j) = 0.0d0
108         do k = 1,n
109           c(i, j) = c(i, j) + a(i, k) * b(k, j)
110         enddo
111       enddo
112     enddo
113
114     return
115     end
116
===== *EOF* =====
```

mmj-after-lno.f

```
166       INTEGER*4 i3
167 C
168 C      **** statements ****
169 C
170       DO j = 1, 200, 1
171         DO i = 1, 200, 1
172           c(i, j) = 0.00000D00
173         END DO
174       END DO
175       DO tile2j = 1, 200, 84
176         DO tile1k = 1, 200, 12
177           DO tile1i = 1, 200, 91
178             DO j0 = tile2j, MIN((tile2j + 82), 199), 2
179               DO k = tile1k, MIN((tile1k + 8), 197), 4
180                 mi0 = b(k, j0)
181                 mi1 = b(k + 3, j0 + 1)
182                 mi2 = b(k + 3, j0)
183                 mi3 = b(k, j0 + 1)
184                 mi4 = b(k + 2, j0 + 1)
185                 mi5 = b(k + 2, j0)
186                 mi6 = b(k + 1, j0)
187                 mi7 = b(k + 1, j0 + 1)
188                 DO i0 = tile1i, MIN((tile1i + 90), 200), 1
189                   c(i0, j0) = (c(i0, j0) +(a(i0, k) * mi0))
190                   c(i0, j0 + 1) = (c(i0, j0 + 1) +(a(i0, k) * mi3))
191                   c(i0, j0) = (c(i0, j0) +(a(i0, k + 1) * mi6))
192                   c(i0, j0 + 1) = (c(i0, j0 + 1) +(a(i0, k + 1) * mi7))
193                   c(i0, j0) = (c(i0, j0) +(a(i0, k + 2) * mi5))
194                   c(i0, j0 + 1) = (c(i0, j0 + 1) +(a(i0, k + 2) * mi4))
195                   c(i0, j0) = (c(i0, j0) +(a(i0, k + 3) * mi2))
196                   c(i0, j0 + 1) = (c(i0, j0 + 1) +(a(i0, k + 3) * mi1))
197                 END DO
198               END DO
199               DO wd_k0 = k, MIN((tile1k + 11), 200), 1
200                 mi8 = b(wd_k0, j0)
201                 mi9 = b(wd_k0, j0 + 1)
202                 DO i1 = tile1i, MIN((tile1i + 90), 200), 1
203                   c(i1, j0) = (c(i1, j0) +(a(i1, wd_k0) * mi8))
204                   c(i1, j0 + 1) = (c(i1, j0 + 1) +(a(i1, wd_k0) * mi9))
205                 END DO
206               END DO
207             END DO
208             DO wd_j = j0, MIN((tile2j + 83), 200), 1
209               DO k0 = tile1k, MIN((tile1k + 8), 197), 4
210                 mi10 = b(k0, wd_j)
211                 mi11 = b(k0 + 3, wd_j)
212                 mi12 = b(k0 + 1, wd_j)
213                 mi13 = b(k0 + 2, wd_j)
214                 DO i2 = tile1i, MIN((tile1i + 90), 200), 1
215                   c(i2, wd_j) = (c(i2, wd_j) +(a(i2, k0) * mi10))
216                   c(i2, wd_j) = (c(i2, wd_j) +(a(i2, k0 + 1) * mi12))
217                   c(i2, wd_j) = (c(i2, wd_j) +(a(i2, k0 + 2) * mi13))
218                   c(i2, wd_j) = (c(i2, wd_j) +(a(i2, k0 + 3) * mi11))
219                 END DO
220               END DO
221               DO wd_k = k0, MIN((tile1k + 11), 200), 1
222                 mi14 = b(wd_k, wd_j)
223                 DO i3 = tile1i, MIN((tile1i + 90), 200), 1
224                   c(i3, wd_j) = (c(i3, wd_j) +(a(i3, wd_k) * mi14))
225                 END DO
226               END DO
227             END DO
228           END DO
229         END DO
230       END DO
231       RETURN
232       END ! mmjik200
```


Figure 1: **SYNCHRONIZING ORIGINAL SOURCE and TRANSFORMED SOURCE: The example we are seeing is matrix multiply (mmjik200) of a two 200x200 matrices. The window on the left is the original source and the window on the right is the transformed source.** (View in color)

```
(LNO_SNL
  (NESTING_DEPTH 3)   (LINE_POS 71 72 74)
  (IF_INNER 74  (CYCLES 1.20959 (1 "Ideal Schedule")  0.114022  0.0955658)
    (FP_REGISTERS 12)
    (TRANSFORMATIONS  (UNTILED_ORDER 71 72 74)
      (UNROLL (71 2) (72 2))
      (BLOCKING (74 132 L2 71) (71 72 L2 71) (72 14 L1 71) (74 66 L1 71))))
  (IF_INNER 72  (CYCLES 1.20219  (1 "Ideal Schedule") 0.154194 0.0479915)
    (FP_REGISTERS 28)
    (TRANSFORMATIONS  (UNTILED_ORDER 71 74 72)
      (UNROLL (71 2) (74 4))
      (BLOCKING (71 84 L2 71) (74 12 L1 71) (72 91 L1 71))))
  (IF_INNER 71  (CYCLES 1.24827  (1 "Ideal Schedule") 0.161221 0.0870466)
    (FP_REGISTERS 28)
    (TRANSFORMATIONS (UNTILED_ORDER 72 74 71)
      (UNROLL (72 2) (74 4))
      (BLOCKING (71 90 L2 72) (72 126 L2 72) (74 20 L1 72) (71 45 L1 72))))
  (INNER_LOOP 72)
)
```


Figure 2: **EXAMPLE OF COMPILER INFORMATION FOR mmjik200 LOOP NEST: This information is produced by the compiler in a separate file that jCITE knows and can read. It has information on various loop optimizations including cache blocking, register blocking, fission/fusion, and prefetching.**

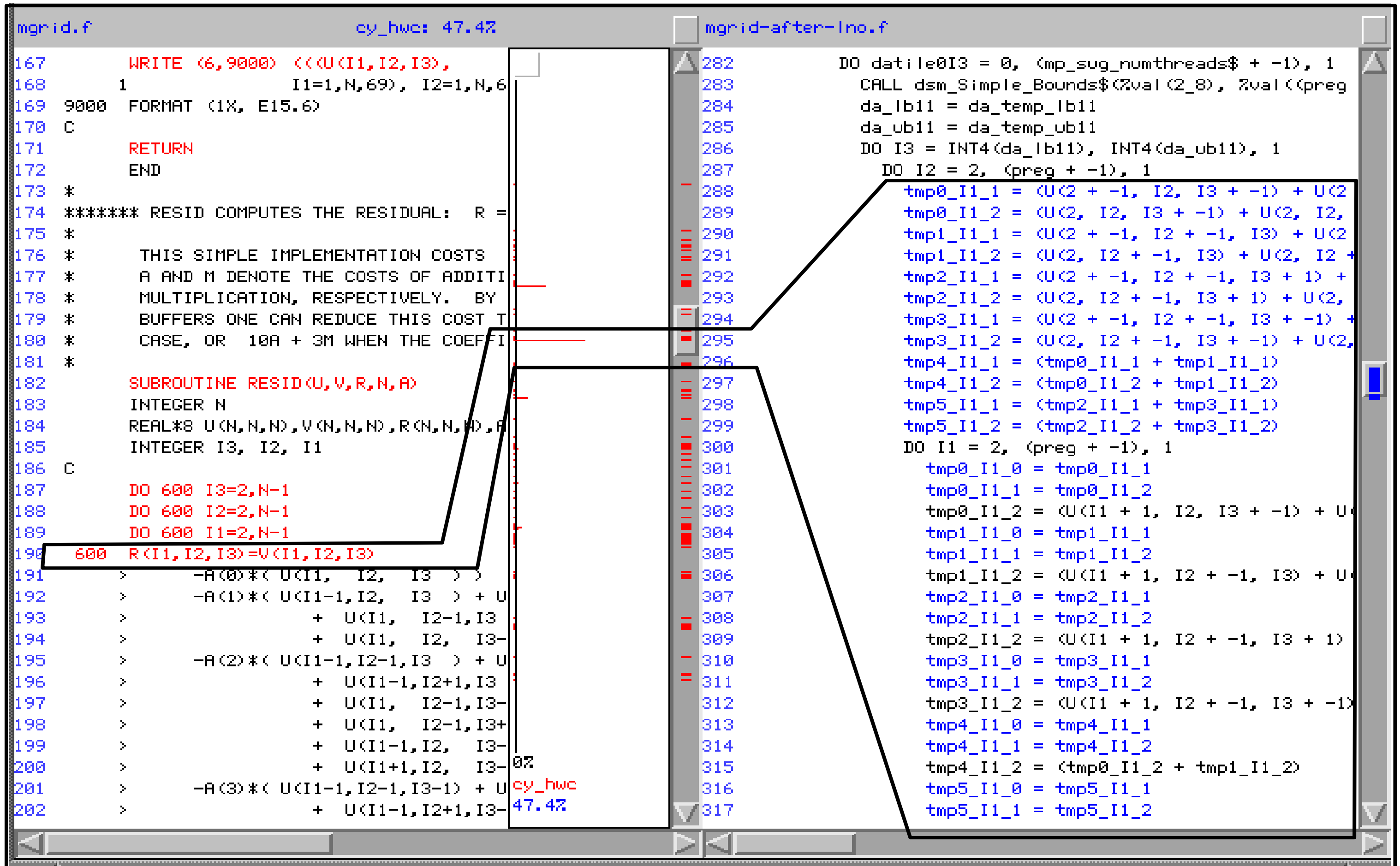

Figure 3: **ANNOTATED PERFORMANCE INFORMATION ON THE SOURCE The example we are seeing is from the SPEC95 fp benchmark 107.mgrid. The left window is the original source annotated with cycle counts from the R10000 hardware counters and parallelization information. A histogram view of the per line cycle counts is overlayed on top of the source window. Navigating the histogram navigates the source and transformed source. The snapshot was taking with the mouse in the histogram window for the most number of cycles (43.4 % of all cycles). This corresponds to the 3D array assignment in the subroutine** `resid`

```
mgrid.f
167        WRITE (6,9000) (((U(I1,I2,I
168       1              I1=1,N,69),
169  9000  FORMAT (1X, E15.6)
170  C
171        RETURN
172        END
173  *
174  ******* RESID COMPUTES THE RESIDU
175  *
176  *      THIS SIMPLE IMPLEMENTATION
177  *      A AND M DENOTE THE COSTS O
178  *      MULTIPLICATION, RESPECTIVE
179  *      BUFFERS ONE CAN REDUCE THI
180  *      CASE, OR  10A + 3M WHEN TH
181  *
182        SUBROUTINE RESID(U,V,R,N,A)
183        INTEGER N
184        REAL*8 U(N,N,N),V(N,N,N),R(
185        INTEGER I3, I2, I1
186  C
187        DO 600 I3=2,N-1
188        DO 600 I2=2,N-1
189        DO 600 I1=2,N-1
190    600 R(I1,I2,I3)=V(I1,I2,I3)
191       >      -A(0)*( U(I1,  I2,  I
192       >      -A(1)*( U(I1-1,I2,  I
193       >               + U(I1,
194       >               + U(I1,
```

```
0%        0%
cy_hwc    dsc_hwc
```

```
mgrid-after-lno.f
282          DO datile0I3 = 0, (mp_sug_numthreads$ + -1), 1
283            CALL dsm_Simple_Bounds$(%val(2_8), %val((preg
284            da_lb11 = da_temp_lb11
285            da_ub11 = da_temp_ub11
286            DO I3 = INT4(da_lb11), INT4(da_ub11), 1
287              DO I2 = 2, (preg + -1), 1
288                tmp0_I1_1 = (U(2 + -1, I2, I3 + -1) + U(2
289                tmp0_I1_2 = (U(2, I2, I3 + -1) + U(2, I2,
290                tmp1_I1_1 = (U(2 + -1, I2 + -1, I3) + U(2
291                tmp1_I1_2 = (U(2, I2 + -1, I3) + U(2, I2 +
292                tmp2_I1_1 = (U(2 + -1, I2 + -1, I3 + 1) +
293                tmp2_I1_2 = (U(2, I2 + -1, I3 + 1) + U(2,
294                tmp3_I1_1 = (U(2 + -1, I2 + -1, I3 + -1) +
295                tmp3_I1_2 = (U(2, I2 + -1, I3 + -1) + U(2,
296                tmp4_I1_1 = (tmp0_I1_1 + tmp1_I1_1)
297                tmp4_I1_2 = (tmp0_I1_2 + tmp1_I1_2)
298                tmp5_I1_1 = (tmp2_I1_1 + tmp3_I1_1)
299                tmp5_I1_2 = (tmp2_I1_2 + tmp3_I1_2)
300                DO I1 = 2, (preg + -1), 1
301                  tmp0_I1_0 = tmp0_I1_1
302                  tmp0_I1_1 = tmp0_I1_2
303                  tmp0_I1_2 = (U(I1 + 1, I2, I3 + -1) + U(
304                  tmp1_I1_0 = tmp1_I1_1
305                  tmp1_I1_1 = tmp1_I1_2
306                  tmp1_I1_2 = (U(I1 + 1, I2 + -1, I3) + U(
307                  tmp2_I1_0 = tmp2_I1_1
308                  tmp2_I1_1 = tmp2_I1_2
309                  tmp2_I1_2 = (U(I1 + 1, I2 + -1, I3 + 1)
```

```
Added info for:     DO 600 I3=2,N-1

Loop at line 187 is PARALLEL:
      DO 600 I3=2,N-1
There are 3 killed scalars: I3, I2, I1
There are 3 def scalars: I3, I2, I1
```

Figure 4: **COMBINING PERFORMANCE INFORMATION and COMPILER INFORMATION: The example we are seeing is from the SPEC95 fp benchmark 107.mgrid. The left window is the original source annotated with cycle counts and secondary cache miss counts from the R10000 hardware counters. The histogram overlays show the overall cycle counts and cache misses. The frame at the bottom is as a result of a query to obtain the compiler optimization information.**

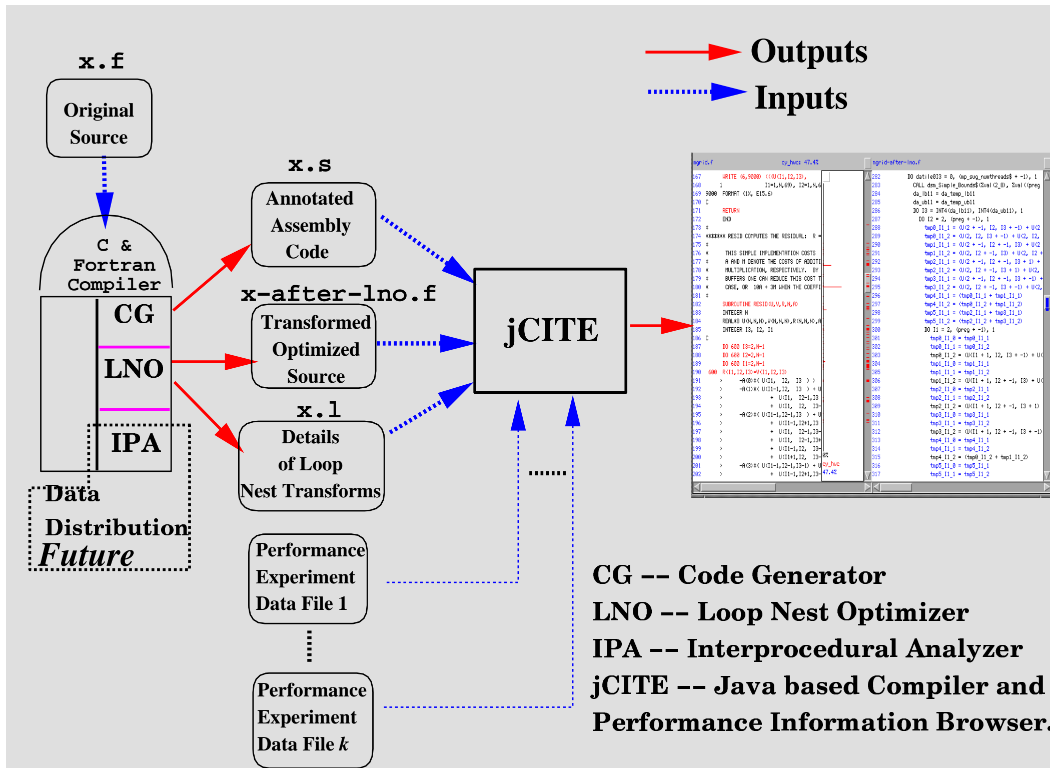

Figure 5: **BLACK BOX ARCHITECTURE: jCITE takes several inputs—the original source, the transformed source, assembly, compiler information file, and performance experiment data files if any. Its output is a visual representation showing how all its inputs relate to one another. For example showing how the source/transformed source relate to one another. At present jCITE works for Fortran 77 and C.**

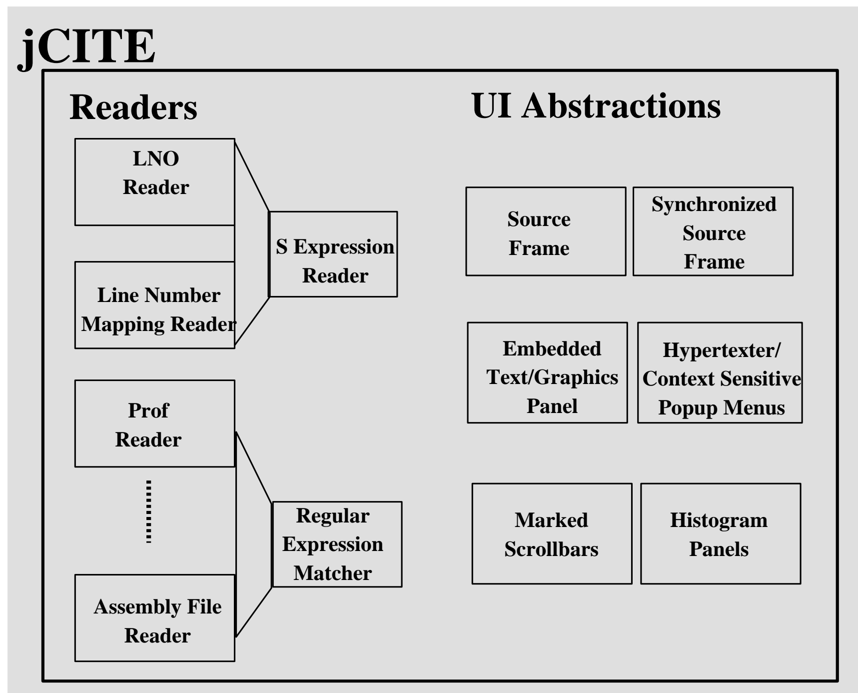

Figure 6: **INTERNALS of jCITE: Two main components, the READERS and the UI ABSTRACTIONS.**