A portable browser for performance programming

MICHAŁ CIERNIAK *
Computer Science Department
University of Rochester
Rochester, NY 14627-0226
cierniak@cs.rochester.edu

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

SUMMARY

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 desktop machine (O2, Octane, Indy) or on a PC running Windows to analyze and optimize the performance of a scientific code running on a SGI Challenge or an SGI Origin machine.

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. A longer version of this paper [1] describes all our experiments with *jCITE*.

1 INTRODUCTION

Performance is at the heart of scientific and engineering computing. Tuning a critical inner loop could lead to significant reduction in the running time of a large scientific or engineering application. This paper is about a portable performance programming tool 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" [2]. 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. Performance programming usually requires manual tuning of the source code and compiler options to obtain the fastest code.

The unfortunate reality is that performance programming is hard. The advent of optimizing compilers which perform various transformations on the program source to utilize

*This work was done while Michał Cierniak worked at Silicon Graphics Incorportated.

the memory hierarchy better has made performance programming even harder. To add to that the scientific and engineering codes themselves have gotten more complex.

We have designed jCITE—A Java based browser—to alleviate the performance programming problem. jCITE helps optimize applications whose performance is 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 hotspots (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 [3, 4] 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.

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 synchronizing 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 following section illustrates how jCITE works using a standard benchmark as an example. We show how jCITE can be used to optimize an application in Section 3. Section 4 concludes and describes related and future work.

2 CASE STUDY

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 FORTRAN 77 code. It is also one of the easiest to parallelize by an automatic parallelizer.

We use it as a case study to illustrate:

- Annotating the application source with runtime performance information on 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.

Figure 1 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.

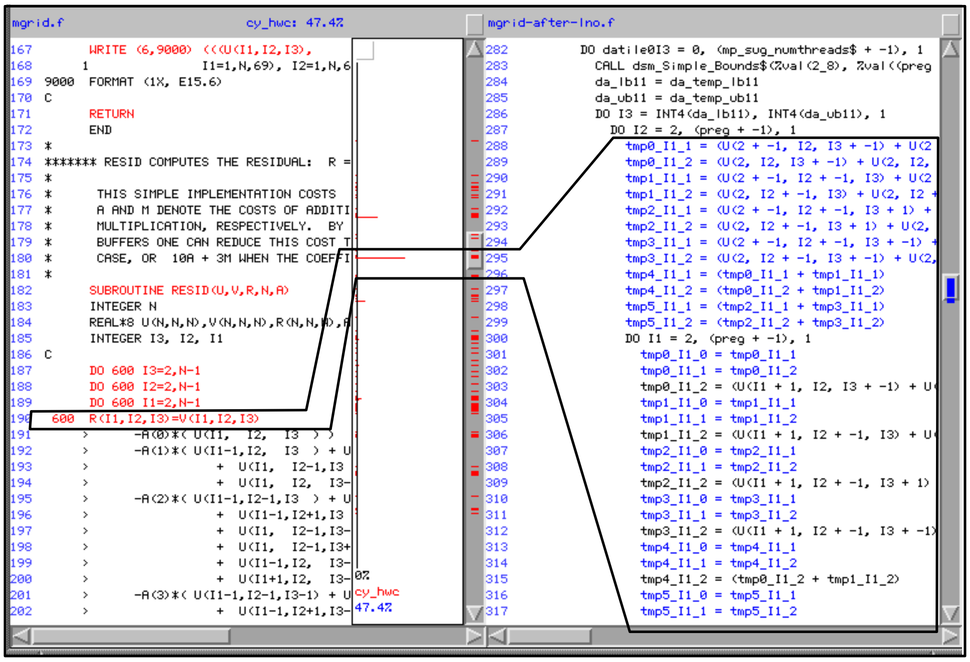

Figure 1: Annotated performance information.

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 over the line with the largest number of processor cycles (43.4 % of all cycles). This corresponds to the 3D array assignment in the subroutine `resid`.

For the loop nest presented 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.

jCITE's architecture is illustrated in Figure 2. 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. At present jCITE works for Fortran 77 and C.

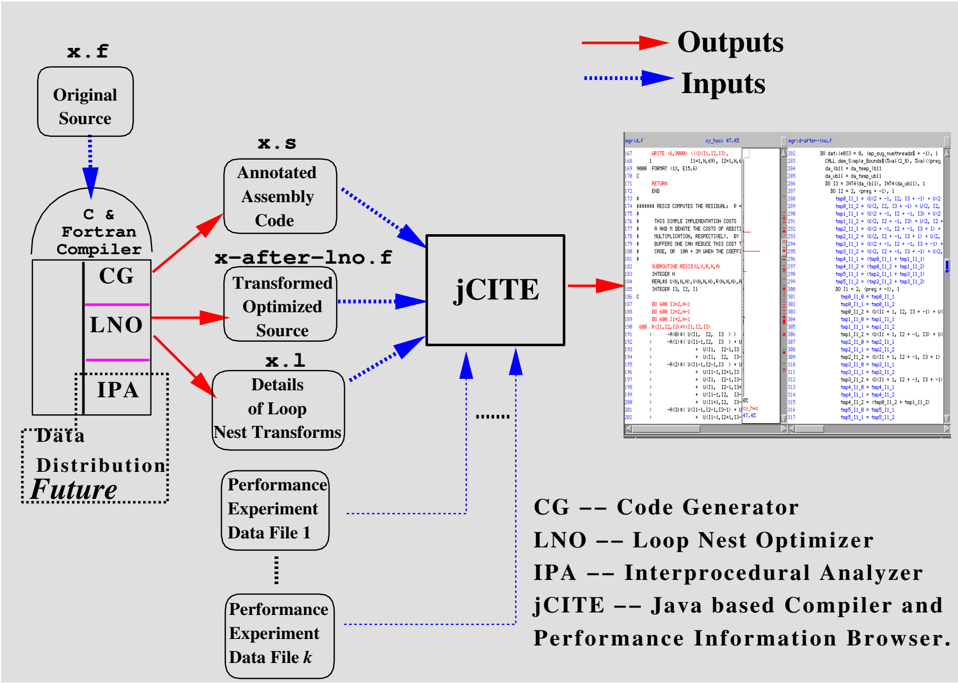

Figure 2: Black Box Architecture of jCITE

3 EXPERIMENTS

In this section we present one of the experiments that we did using jCITE. For other examples in the scientific and engineering domain refer to the longer paper [1].

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.

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`). The details of various optimizations we did can be found in a longer version of this paper [1].

4 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 application performance matches the hardware.
- Novel visual displays are much more attractive in presenting performance data than static text data. Correlating data to provide information is even more useful.
- 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 [5] is 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 used by us has been insufficient for programs such as jCITE that read many files.
  - The lack of features in AWT (JDK 1.0.2) such as pop-up menus or 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.
- 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 its dependence on XEmacs.

4.1 Related work

Sigma [7] 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 Parascope programming environment, the D editor [8] 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 [9] provides information on KAI parallelizer optimizations. It works only on SGI computers. KAI [10] 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.

Acknowlegements

We would like to acknowledge the members of the "Developer Magic Group" working on Compilers and Tools at Silicon Graphics Inc. for all their help. Special thanks to Ron Price and Dror Maydan for their encouragement. The first author was funded through a summer internship when much of this work was completed.

References

[1] Michal Cierniak and Suresh Srinivas. A Portable Browser for Performance Programming. Available at `http://reality.sgi.com/ssuresh_engr/papers.html`.

[2] 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`.

[3] 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`.

[4] Silicon Graphics, Inc. MIPS R10000 Performance Counters. Available at `http://www.mips.com/products/r10k/Perf_Cnt`, March 1997.

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

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

[7] 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.

[8] 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`.

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

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