ArrayTracer: A Parallel Performance Analysis Tool

Titos Saridakis (*sarid@ics.forth.gr*) Christos Nikolaou (*nikolau@ics.forth.gr*)
Maria Karavassili (*karma@ics.forth.gr*) Evangelos Markatos (*markatos@ics.forth.gr*)

Institute of Computer Science – FORTH
Heraklion, Crete, Greece

November 1995

Abstract

ArrayTracer is a high–level, low–overhead performance analysis tool for parallel applications. It provides the selective tracing facilities at a user–defined grain. The tracing technique used is *program instrumentation*. Instrumentation code is inserted at source code level during a source–to–source translation. The tool allows tracing of application's high–level concepts (e.g. program variables, subroutine calls, interprocess communication events, etc.) rather than tracing of low–level events (e.g. memory location accesses, disk accesses, network accesses, etc.). *ArrayTracer* is heavily based on compile–time information and attempts to off–load the execution time tracing overhead by extracting as much information as possible during the *static analysis* of application's source code. The tool's operations are divided into five stages: user interaction to determine *trace parameters*, instrumentation of source code during the compilation phase, collection of traces, processing of traces and analysis of the application's behavior, and presentation of the analysis results. In this report we focus on the first three stages, and present an encouraging preliminary performance evaluation.

Keywords: *tracing, program distortion, program dilation, program instrumentation, selective tracing*

1 Introduction

Most large–scale high–performance parallel application codes are inherently complex and have a non–deterministic nature caused by the concurrent execution of different program components. As a result, effective tools are necessary to assist users in understanding program performance and run–time behaviour. These tools are usually referred to as *Parallel Performance Analysis Tools* (PPA tools). The operation of today's PPA tools is decomposed into three main functional levels [10]: trace collection, trace processing, and result visualization.

Considerable amount of work has been done in all three functional levels since each one has to deal with important issues. Trace collection should be optimized as not to perturb the execution of the application while collecting all the desirable information requested by the user. Trace processing should be clever enough to distinguish those events that cause performance problems among all traced events. The visualization level has to deal with the presentation of large volumes of result data that may be produced by massively parallel applications.

The trace collection issue is not a recent one. A number of user–tools are heavily based on traces, including correctness debuggers [4, 15, 24, 14, 20, 13], performance debuggers[8, 9, 17, 16, 7], trace driven simulators [2, 26, 5, 22, 3], etc. A broad classification [23] of tracing techniques distinguishes four basic classes: *hardware–based* methods which use a hardware monitor to record all requests on the address bus of a processor, *interrupt–based* methods which cause an interrupt on every instruction that accesses some memory location [1], or that misses a simulated memory structure [19, 25], *microcode–based* methods (used only in microcode–based processors) where traces are produced by appropriately modified microcode [21], and *instrumented program–based* methods that introduce tracing statements in application's code which are responsible for producing the traces; instrumentation code may be inserted before the compilation of the application [27, 18, 7], during the compilation [2, 5] or even during the execution of the application [12]. Hardware–based methods are characterised by a low–degree of intrusiveness which results in low overhead and low perturbation of the execution of the target application. However, their cost, inherent inflexibility and inability to provide high–level monitoring information limit their applicability in application–dependent monitoring. Software monitoring or hybrid software–hardware tracing methods are used by the majority of existing PPA tools.

In this paper we present *ArrayTracer*, a high–level low–overhead performance analysis tool for parallel applications written in FORTRAN, using PVM for communication. The tracing technique used within *ArrayTracer* has two major characteristics that distinguish it among others tools in the field of Parallel Performance Analysis: It uses *selective instrumentation* to application source code which is assisted through use of a small, yet flexible set of tracing specification commands. In this manner, the user can tune the tracing process both at the level of program events which should be traced and at the level of code regions where the tracing process should take place. The tool collects traces concerning high–level source code events rather than low–level run–time environment's events, so no additional mapping is required [11] to present the information included in the traces to the user. Therefore, *ArrayTracer* allows the user to orientate the tracing process and focus it on selected parts of the application. Moreover, it facilitates the analysis of the application's behavior in terms of its internal structure (source code concepts) rather than in terms of the characteristics of the given system on which the application is run. Thus, the user is facing the problems in the application's design instead of problems emerging in the specific environment where the program is run. This test focuses on the first three stages in *ArrayTracer* 's operation: user interaction to determine the tracing parameters, instrumentation of source code during the compilation phase and collection of traces.

The rest of this paper is structured as follows: Section 2 gives an overview of *ArrayTracer*. Experiments with *ArrayTracer* and a brief performance evaluation of our tool in the area of PPA tools is given in section 3. Our conclusions from the experimentation with *ArrayTracer* are summarised in section 4.

2 Overview

Trace collection in *ArrayTracer* is performed in three stages. Initially the user determines the trace parameters during the *preparation stage*. In the *instrumentation* or *static analysis* stage that follows, the application code is instrumented. Finally the *trace collection* stage is responsible for collecting the traces and forwarding them in the trace processing module. Figure 1 shows the flow of instrumented code and trace data within *ArrayTracer* . These three stages are discussed next.

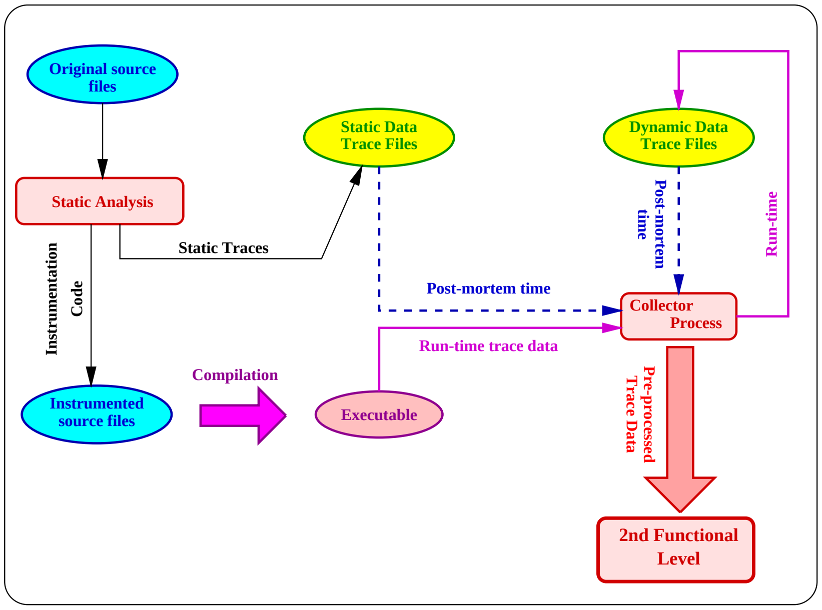


Figure 1: Abstract flow of instrumentation code and trace data in *ArrayTracer* .

Preparation Stage. In this stage, the user determines the source code concepts that should be traced, which will be hereafter called *trace parameters*. These include: program variables (scalar variables, arrays and sub−arrays), source code basic concepts (loops, function/subroutine calls, etc.), and interprocess communication events (message send/receive, barriers, etc.). Trace parameters are specified by issuing commands to *ArrayTracer*'s input interface which is actually an interpreter over a small yet powerful set of trace commands. Figure 2 gives an example of some trace commands:

The trace parameters specified are stored into a structure called *trace table*. This structure is a list where each node corresponds to a source code file containing tracing parameters. Each node in this list contains three sublists, one for each category of tracing parameters (variables, code and communication). The *trace table* is used throughout the life−cycle of the tool. It is created during the *preparation stage* and it is completed during the *static analysis stage* where information regarding the dimensions of arrays and the grouping of variables according to the FORTRAN *EQUIVALENCE* and *COMMON* declarations is

trace file */users/Mike/my_appl.f*	Specify the source−code file
trace variable *a* **from** *30* **to** *150*	Specify scalar variable
trace variable *c[1..10]* **from** *100* **to** *300*	Specify sub−array
trace code at loop level from *50* **to** *125*	Specify tracing of loops
trace code at subroutine level from *100* **to** *200*	Specify tracing of sub−routine calls
bye	Quit the tool

Figure 2: Example of user instructions in the preparation stage.

available. Upon termination of the specification of trace parameters for a given file, the newly constructed *trace table* node is sent to the *static analysis* module. Thus, it it possible for the user to specify trace parameters for a source file, while static analysis is being performed to the file previously marked for tracing, which speeds up the tracing process. The *trace table* is also used in the trace collection stage and the trace processing at post−mortem time as discussed later.

Static Analysis Stage. During this stage, *ArrayTracer* parses the application's source code, identifies all references to trace parameters (according to information stored in the *trace table*) and inserts appropriate instrumentation code. The tool classifies all program events into two categories: those that interest the user (they are marked for tracing during the preparation stage), and those that do not interest the user. However, in some cases one cannot tell whether a given program event is related with some trace parameter[1]. In those cases, *ArrayTracer* inserts instrumentation code and later (during the processing of the traces) it decides whether the traces produced interest the user − *essential traces* − and should thus be used for the analysis of the application's behavior, or not, in which case they are characterised as *non−essential traces*. The output of this stage is a file containing the original application's source code augmented with the calls to tracing routines. Figure 3 illustrates the instrumentation process. We assume that variables *M* and *N* take their values at run−time.

The module implementing this stage is a *source−to−source* compiler which results in making *ArrayTracer* independent from any other tools and thus highly portable to other machines. Additionaly, it facilitates the debugging process since we can view the instrumented file. In the current implementation, we do not collect any traces available during the static analysis time, instead we insert a call to a trace routine and defer their production at run−time.

Inserting instrumentation code in *FORTRAN* code is a straightforward process of corresponding program events to trace parameters, except from two cases: equivalent variables[2] and tracing inside subroutines bodies. When one variable in a set of equivalent variables is marked for tracing, all variables in

[1]Such cases are when the borders of a certain array block being accessed are determined during run−time, so one wouldn't know whether the given array block intersects with some sub−array marked for trace or not.

[2]In FORTRAN, when two variables are declared in an *EQUIVALENCE* (or *COMMON*) declaration, they occupy the same memory location.

User Input:

trace variable *a[1..10]* **from** *1* **to** *100*
trace variable *b* **from** *1* **to** *50*

```
Original Source Code:           Instrumented Source Code:
INTEGER i, a(100), b, c         INTEGER i, a(100), b, c

DO i = M, N                     DO i = M, N
   a[i] = c * b                    a[i] = c * b
                                   CALL tracevar(a_TAG, i, OP_WR)
                                   CALL tracevar(b_TAG, 0, OP_RD)
   b = c + 1                       b = c + 1
                                   CALL tracevar(b_TAG, 0, OP_WR)
   c = c + 5                       c = c + 5
ENDDO                           ENDDO
```

Figure 3: Selective insertion of instrumentation code.

this set should be traced too and so they are inserted into the trace table. A problem occurs when sub−blocks of two arrays (possibly with different number of dimensions and different sizes) are declared equivalent. In this case, the equivalent elements of the two arrays are identified and the corresponding sub−blocks are inserted in the trace table. Tracing inside a subroutine's body is performed when the given subroutine is called with an argument which is a trace parameter. In that case, the subroutine's body should be parsed and appropriate instrumentation code should be inserted. However, this does not suffice[3] so a copy of the subroutine is made every time the parser finds a call to the subroutine with arguments related to some trace parameter. Appropriate instrumentation code is inserted to these copies too.

Trace Collection Stage. During this stage, *ArrayTracer* creates and attaches a *collector*−process that collects the traces produced to each running application process. This process runs on the same CPU processor with the application process to which it is attached on and communicates with it via a shared buffer. At the static analysis stage, the source−to−source compiler inserts also instrumentation code responsible for creating the shared buffer. This module attaches the shared buffer to the application process memory space, starts the collector−process and passes to it information on how to access the shared buffer. Instrumentation code is also inserted immediately before the point where the application process exits. This code is responsible for informing the collector−process about the application process termination.

During the execution of the application, the *collector*−process copies blocks of the shared buffer into a trace−file on the disk, which will be processed at post−mortem time. Using a collector process and a shared buffer for storing traces on the disk is transparent to the design of the tracing technique used by *ArrayTracer* . It can be changed if necessary by another block−writing mechanism, without any side−effects on our tool. Once the application process terminates, the *collector*−process is responsible for retrieving the

[3]A subroutine may be called several times in a program and each time with different arguments which all might be trace parameters, so the traces that should be produced at each call should be different.

stored traces from disk, separating the *essential* from the *non−essential* traces and sending only the former to the *ArrayTracer*'s core−module for the trace processing phase.

3 Performance Results

In this section we describe the experiments conducted to evaluate *ArrayTracer*'s performance. The current implementation of *ArrayTracer* runs on *DEC−ALPHA 3000* workstations under *DEC OSF/1 V3.0 Worksystem Software*. The code is written in *C* and *gcc version 2.7.0* is used. The communication of the tool's modules is done over *PVM version 3.3.2*. Current implementation includes: the Input Interface, the pre−processor responsible for the static analysis, the shared buffer mechanism, and the collector−process. A graphical user interface for the visualization of the trace processing results is under construction.

Execution slowdown due to *ArrayTracer*: We run various experiments in order to measure the *dilation*[4] that *ArrayTracer* introduces to the application it traces. The applications used were a sequential and a parallel implementation of a smoothing algorithm and a chemistry application for the calculation of the energy trajectories of electrons in the H_2O molecule. All these applications read their input from disk and write their output back to disk. In these cases, the dilation metric was measured to be very low (around 2 to 7). In figure 4, the application named *200 parallel* indicates the parallel version of the smoothing algorithm

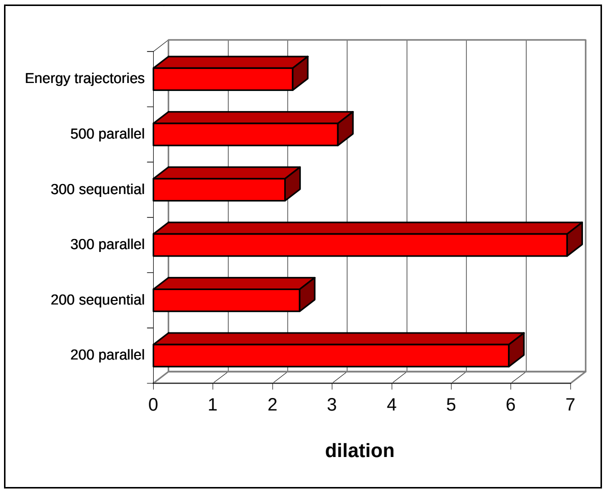


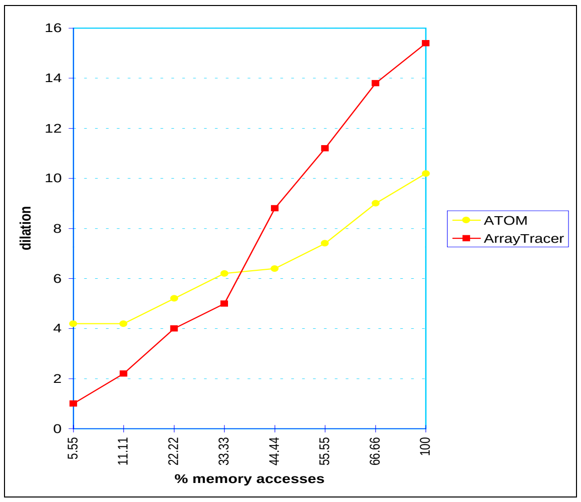
Figure 4: Mesuring *ArrayTracer*'s dilation.

for an array with size 200x200. Parallel smoothing executes on four slave processes which sent results to a master process to save them on disk. The *500 parallel* application runs on five machines. The rest of the parallel runs where done in a single CPU. This is the reason for the dilation factor near 7. The runs for the energy trajectories application were also done in a single CPU and it used one master process and two slave

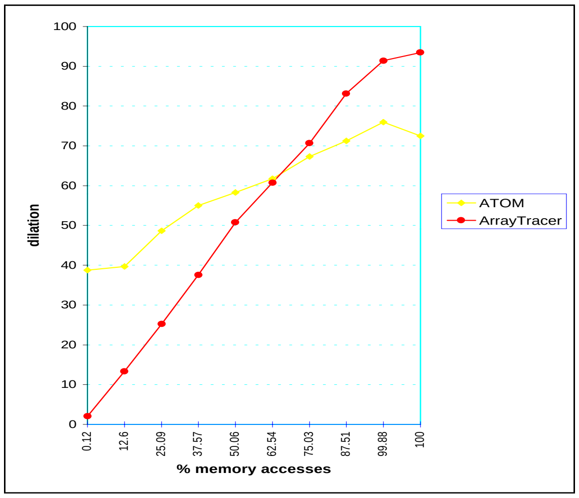
[4]Dilation represents the overhead tracing imposes to the executed application and is usually measured as the ratio of the completion time of an application with tracing to the completion time of the same application without tracing. Dilation factors around 2−3 are considered excellent, while dilation factors around 1000 are considered very bad.

processes.

Trace−driven simulation vs. selective tracing: To evaluate the benefits of selective tracing we have compared *ArrayTracer* with *ATOM* [22], a fast tracing tool that cannot selectively trace only a few variables. Results are very encouraging when the *selective tracing* facility of the *ArrayTracer* is used to bound the tracing process in a given set of program events. We have programmed *ATOM* to collect traces only for some selected program events too. The applications used were the sequential version of the smoothing algorithm, a matrix multiplication and an application which calculates the position of a point in the 3D space after a predefined sequence of movements[5]. Note that, while for *ArrayTracer* the specification of tracing parameters was straightforward, when using *ATOM* we had to find the virtual memory addresses that correspond to variables that should be traced and then collect traces for these addresses. The results of this comparison for the smoothing and the matrix multiplication algorithms, are shown in figure 5.



(a) Smoothing algorithm.



(b) Matrix Multiplication.

Figure 5: *ArrayTracer* vs. *ATOM*

Looking at figure 5 one may see that when the user marks for tracing less than 35% of variable references in the application, *ArrayTracer* behaves better than *ATOM*. Especially, in the matrix multiplication[6] we notice that *ArrayTracer* performs better when tracing up to 65% of variable references. The difference in the crossing points in 5(a) and 5(b) happens for the following reason: *ArrayTracer* traces accesses to *program variables* whereas *ATOM* traces accesses to *memory locations*. Thus, if a certain variable is placed into a register, *ATOM* does not trace accesses to that variable since they are not memory accesses, while *ArrayTracer* traces the accesses to that variable. In the smoothing algorithm, there were some frequently

[5]A movement includes rotation around a random axis and translation towards a random direction.

[6]In this figure we notice that *ATOM*'s elapsed time is somewhat lower when tracing 100% of variable accesses than it is when tracing 99.88%. This happens because the instrumentation code inserted by *ATOM* checks every load/store operation to find if the memory address specified in it belongs to the given range that the user had selected for tracing, which becomes complex when tracing a large percentage of load/store operations, as in the case of 99.88%. However, when *ATOM* traces 100% of variable references there is no need for this checking, which results in lower elapsed time.

used variables placed into registers and thus less traces were produced under *ATOM*. However in the matrix multiplication, most accesses were to matrices which cannot be placed in registers. Therefore, *ATOM* had to produce as many traces as *ArrayTracer* did.

These performance results are particularly encouraging given that in most cases users want to focus on a small portion of code and data of parallel applications in order to detect performance and correctness flows. *ArrayTracer* is particularly effective when it traces only a small portion of the data. For example, when only 5% of the data are traced, figure 5(b) shows an order of magnitude improvement over other state–of–the–art tools, like *ATOM*. We believe that we will observe similar behaviour if we compare *ArrayTracer* with communication event tracing tools like PICL [6]. The ability of *ArrayTracer* to selectively trace communication events, will most probably result in significant performance improvement.

4 Conclusions

We have presented a preliminary introduction and performance results of *ArrayTracer*, a parallel per– formance analysis tool which uses selective instrumentation to produce traces of application source code. There is on–going work on the trace processing and the result visualisation stages. Our experimentation with *ArrayTracer* and other tracing tools such as *ATOM*, led us to the following conclusions:

1. *ArrayTracer* is easy to use and very efficient when the mechanism for selective tracing is used but its performance deteriorates when a large percentage of the program variables are marked for tracing.

2. The dilation introduced by our tool is analogous to the percentage of variables that are traced and thus its behaviour is predictable in contrast to the behaviour of other tools that depend on the compiler and the optimization level used.

3. While other tracing tools that introduce instrumentation code at run–time may cause an unpredictable inflation to the size of the executable, *ArrayTracer* always produces executable files with an anticipated inflation in size (according to the amount of program events specified for tracing).

References

[1] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. "Simple but Effective Techniques for NUMA Memory Managment". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 19––31, December 1989.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. "Proteus: A High–Performance Parallel Architecture Simulator". In *Proceedings of the 12th ACM SIGMETRICS and PERFORMANCE '92 Conference*, June 1992.

[3] B. Cmelik and D. Keppel. "Shade: A Fast Instruction−Set Simulator for Execution Profiling". *ACM SIGMETRICS*, May 1994.

[4] CONVEX Press, Richardson – Texas, USA. *"CXdb Reference: Concepts and Messages"*, 1st edition, November 1992.

[5] H. Davis, S. R. Goldschmidt, and J. Hennessy. "Multiprocessor Simulation and Tracing Using Tango". In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[6] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. *"PICL – A Portable Instrumented Communication Library"*. Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[7] A. H. Goldberg and J. L. Hennessy. "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications". *IEEE Transactions on Parallel and Distributed Systems*, 4(1), January 1993.

[8] J. K. Hollingsworth and B. P. Miller. "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems". Technical Report 1133, University of Madison−Wisconsin, 1993.

[9] J. K. Hollingsworth, B. P. Miller, and J. Cargille. "Dynamic Program Instrumentation for Scalable Performance Tools". Technical Report 1207, University of Madison−Wisconsin, 1993.

[10] A. Hondroudakis. "Performance Analysis Tools for Parallel Programs". EPCC, The University of Edinburgh. Personal communication, July 1995.

[11] R. B. Irvin and B. P. Miller. "A Performance Tool for High−Level Parallel Programming Languages". Technical Report 1204, University of Madison−Wisconsin, 1993.

[12] R. E. Kessler, A. Borg, and D. W. Wall. "Generation and Analysis of Very Long Address Traces". In *Proceedings of the 17th International Conference on Computer Architectures*, 1990.

[13] T. J. LeBlanc and J. M. Mellor−Crummey. "Debugging Parallel Programs with Instant Replay". *IEEE Transactions on Computers*, C−36(4), April 1987.

[14] J. May and F. Berman. "Panorama: A Portable, Extensible Parallel Debugger". *ACM SIGPLAN*, 28(12), December 1993.

[15] B. P. Miller and J.−D. Choi. "A Mechanism for Efficient Debugging of Parallel Programs". Technical Report 754, University of Madison−Wisconsin, 1988.

[16] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.−S. Lim, and T. Torzewski. "IPS−2: The Second Generation of a Parallel Program Measurment System". *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[17] B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan. "The Paradyn Parallel Performance Tools and PVM". Technical Report 1240, University of Madison–Wisconsin, 1994.

[18] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. G. Noe, K. A. Shields, and B. W. Schwartz. "An Overview of the Pablo Performance Analysis Environment". Technical report, University of Illinois, Computer Science Department, 1992.

[19] S. Reinhardt, M. Hill, and J. Larus et. al. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers". *SIGMETRICS 93*, pages 48––60, May 1993.

[20] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. "A Scalable Debugger for Massively Parallel Message–Passing Programs". *IEEE Parallel & Distributed Technology*, Summer 1994.

[21] R. L. Sites and A. Agarwal. "Multiprocessor Cache Analysis Using ATOM". In *Proceedings of the 15th International Conference on Computer Architectures*, 1988.

[22] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools". *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 29(6), June 1994.

[23] C. B. Stunkel, B. Janssens, and W. K. Fuchs. "Address Tracing for Parallel Machines". *IEEE Computer*, 24(1):31––38, January 1991.

[24] M. Timmerman, F. Gielen, and P. Lambrix. "High Level Tools for the debugging of Real–Time Multiprocessor Systems". *ACM SIGPLAN*, 28(12), December 1993.

[25] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. "Tapeworm II: A New Method for Measuring OS Effects on Memory Architecture Performance". In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 132––144, San Jose, CA, 1994.

[26] J. E. Veenstra and R. J. Flower. "MINT Tutorial and User manual". Technical Report 452, The University of Rochester, Computer Science Department, June 1993.

[27] W. Williams, T. Hoel, and D. Pase. "The MMP Apprentice Performance Tool: Delivering the Performance of the Cray T3D". In *Programming Environments for Massivelly Parallel Distributed Systems*, 1994.