A Portable Dynamic Profiler for C++ based Languages

Bernd Mohr
Department of Computer and Information Science
University of Oregon
Eugene, Oregon 97403

email: mohr@cs.uoregon.edu

Abstract

Program profiling is the first step in program tuning. For sequential languages, the necessary profiling code is automatically inserted by the compiler via a command line switch. However, not all compilers for parallel languages have this feature. This article introduces a portable dynamic profiler for (parallel) languages which are based on C++. The first part describes the general approach which allows portable and dynamic program profiling. The second part describes an instrumentation tool for pC++ which we implemented for inserting the necessary profiling code in the program source.

1 Introduction

In general, a very valuable tool for program tuning is function profiling. For function profiling, special code has to be inserted at all entry and exit points of each function. This code is then used to calculate the number of times this function is called and the percentage of the total runtime which was spent in that routine. The necessary computations can be carried out in two ways. First, the profile can be directly computed during the runtime of the program (*direct profiling*). Second, all entry and exit points of a function can be traced and the necessary computations are then done on the basis of the resulting event trace (*trace-based profiling*). The second method has the advantage that the data in the event trace can also be used for further program analysis, e.g., program visualization or animation.

Compilers for sequential languages, especially on UNIX based operating systems, often include a command line option, which automatically instruments the program source for profiling tools like prof or gprof [GKK82]. However, almost all compilers for parallel languages lack this feature, as they are often research tools and emphasis is put on a proper language design and not on supporting performance tools.

In this article we introduce a portable dynamic profiler for (parallel) languages which are based on C++. All necessary program transformations are performed on the C++ language level. This makes our approach extremely portable and flexible. It supports both direct and trace-based profiling. The first part of this article describes how features of C++ can be used to easily instrument the program source for profiling. The next section explains how this can be used for dynamic profiling, i.e., the amount of profiling data which is recorded changes during runtime depending on static and dynamic program properties. This feature is very useful for large parallel systems where it is important to be able to limit the total amount of data to record. The second part describes an instrumentation tool for pC++ (a object-parallel extension to C++) we implemented for inserting the necessary profiling code in the program source. Finally, the current pC++ profiling and tracing environment is described.

2 Profiling C++

This section describes our general approach for profiling programs written in C++ or C++ based languages.

2.1 General Approach

All necessary program transformations are carried out on the language level. This ensures portability. Profiling means inserting special code on all entry and exit points of all functions of a program. As there is no distinction between declarations and executable statements in C++, we simply can add the necessary code for recording the entry into a procedure by inserting it as the first statement of the function. Function exits are more problematic: First, there can be more than one exit point. Second, consider the following small example (see Figure 1).

```
if ( something )
    return ( foo() );

if ( something ) {
    foo_type f = foo();
    code_for_exit();
    return ( f );
}
```

FIGURE 1. Instrumentation example

We have to ensure that the exit code is executed as late as possible. A return expression in general can be an arbitrary (and therefore complex) expression with function calls, and take a very long time. So, we have to extract the expression from the return statement, and compute its value in advance. Then we can execute the profiling exit code, and finally return the computed value of the expression.

Actually, we don't have to do this transformations by ourself. By using a well-known C++ trick, we can let the C++ compiler do the dirty work. The trick is very simple: We declare a special class `Profiler` (see Figure 1) which only has a constructor and a destructor and no other members. If we declare a variable of that class type as an automatic object in the first line of each function which has to be profiled, it is initialized each time the control flow reaches its definition (and the constructor is called) and destroyed on exit from its block (calling the destructor).

```
class Profiler {
    char* name;
public:
    Profiler(char *n) { name=n; code_for_enter(name); }
    ~Profiler() { code_for_exit(name); }
};

void bar()
{
    Profiler tr("bar"); // Profiler variable
    // body of bar
}
```

FIGURE 2. Class Profiler

The C++ compiler is clever enough to rearrange the code by itself and to insert calls to the destructor at the right places. Note also, that we use a private member to store a function identification which we can use in the destructor.

2.2 Dynamic Profiling

The approach described above has basically two advantages:

(1) Instrumenting at the source code level makes it very portable.

(2) Different implementations of the profiler can be easily implemented by providing different code for the `code_for_enter` and `code_for_exit` functions. This makes it very flexible.

Currently, we have implemented two versions of the profiler:

(1) **Direct Profiling**: The function profile is directly computed during the runtime of the program. In order to achieve that, we maintain a set of data values for each function to be profiled. The declaration of the structure to hold these values is shown in

```
struct p_func_descr {
    int numcalls;        // number of calls
    double usec;         // microseconds spent in this
                         // function alone
    double cumusec;      // microseconds spent in this
                         // function and all its childs
}
```

FIGURE 3. Structure for holding Function Profile Data

Figure 1. The actual implementation of the direct profiler uses two additional members compared to the Profiler class declaration defined in Figure 1. One for storing the timestamp of the entry into the function and the second to remember the parent function (i.e., the function who called the current function). Both members are set by the constructor (`code_for_enter`), which also increments the `numcalls` field. The destructor (`code_for_exit`) uses the entry timestamp to compute the duration of the function call and adds this value to the corresponding `usec` and `cumusec` fields, but also subtracts it from the `usec` field of its parent function. Through this trick we can compute the time spent in a function itself not counting its children. If there is no parent function, i.e., the function is the main function, the profile data gathered for all functions is written to a file by the destructor.

(2) **Trace-based Profiling**: Here the `code_for_enter` and `code_for_exit` functions simply call an event logging function from the portable software event tracing module [mohr93b]. All events inserted are assigned to the event class EC_PROFILER. By using event classes, the event recording can be activated/ deactivated at runtime (see *se_event*(3)). The computation of the profile statistics is then done off-line.

Other alternatives could be implemented in the same way. For example, there could be an implementation where the profiling code can be activated/deactivated for each function separately. One way of achieving this would be to have a global array with an entry per function, which indicates what to do when this function is called (e.g., do not profile, profile, stop execution, etc.) This array could be initialized at runtime making it a very flexible and dynamic profiling environment which can be controlled by the user. Yet another alternative would be to have user supplied, function specific profile code (e.g., specified through source code annotations or special class members with predefined names).

3 Instrumenting pC++

pC++ is an object-parallel extension to the C++ programming language [BBG+92b, BBG+93, gann93]. The pC++ compiler (*pc++*) consists of several stages which are shown in Figure 4. pC++ source code is first parsed using *pC++2dep*, one file at a time to produce a machine independent binary internal format called a *.dep* file. The program *dep2C++* transforms it into a SPMD style C++ program for a parallel computer system. The C++ file is then compiled with the help of a conventional C++ compiler (like CC or g++) and linked together with the pC++ runtime system to an executable.

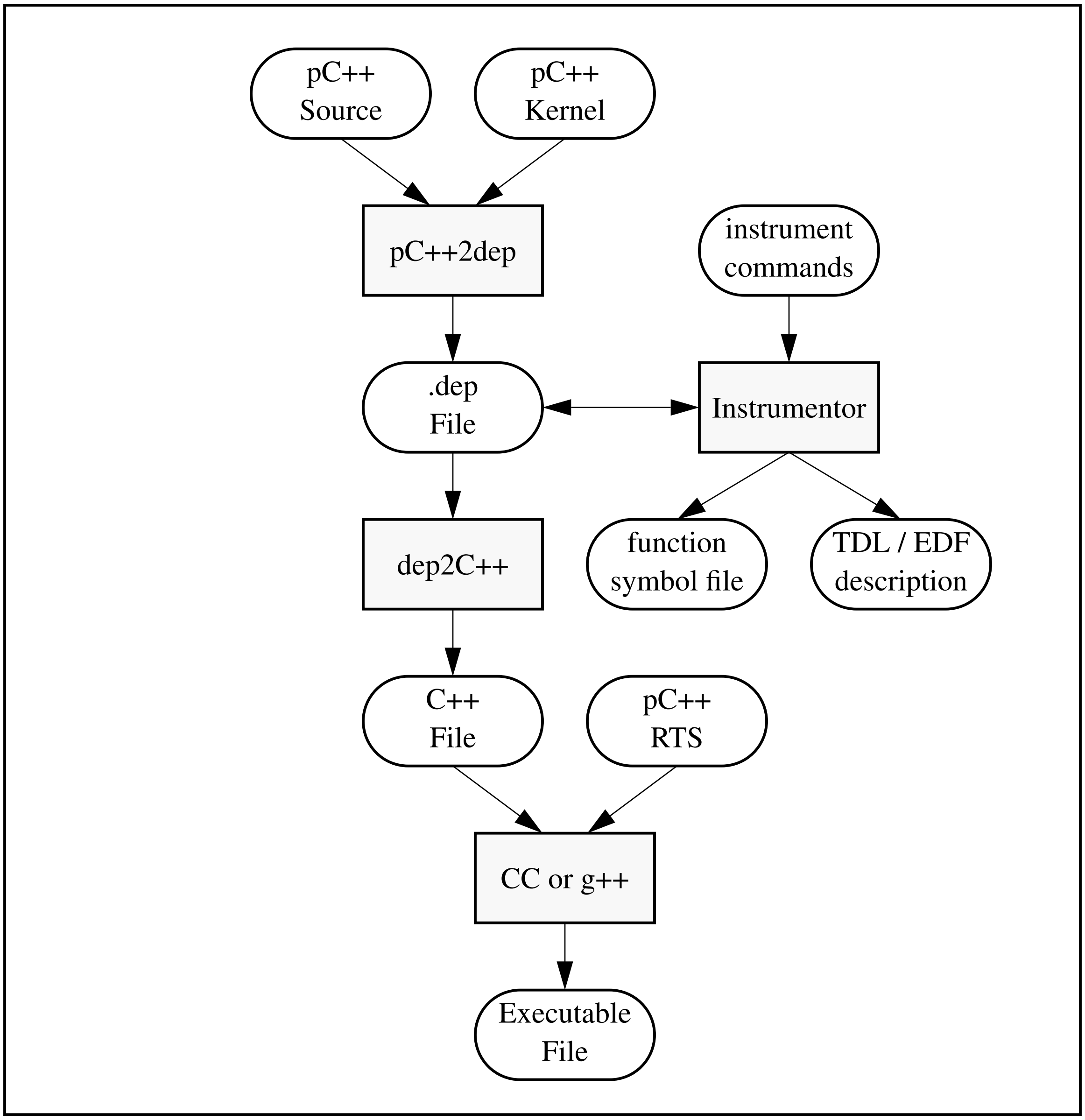

FIGURE 4. pC++ Translation Process

The *dep2C++* translator is implemented on top of *Sage++*, a sophisticated class library for building C++ restructuring tools [BBG+92a].

3.1 Implementation of the Instrumentor

As our approach for profiling is based on source code instrumentation, we can use the *Sage++* library for manipulating the pC++ program source in order to insert the necessary profiler class variable declarations at the beginning of each function. So, the instrumentor consists basically of three phases: (1) read the corresponding .dep file, (2) manipulate the syntax tree with the help of Sage++ according to commands in a so-called *instrumentation command file*, and (3) write the changes back into the .dep file.

The instrumentor supports both *direct* profiling and *trace-based* profiling. On demand, the instrumentor can produce description files which are helpful for analyzing the resulting profile data. For direct profiling, a function symbol file (named "`profile.ftab`") is generated which is used by the parallel profile printing tool (*pprof*). For trace-based profiling, a TDL description [mohr90] of the profiling trace file and a general event description file (EDF) is generated.

3.2 The Instrumentation Command File

By default, every function in the given C++ input files is profiled. However, the user can specify the set of functions to instrument with the help of a instrumentation command file. The file can contain a sequence of instrumentation commands, one per line. Empty lines are ignored. The # character introduces a comment terminated by a newline.

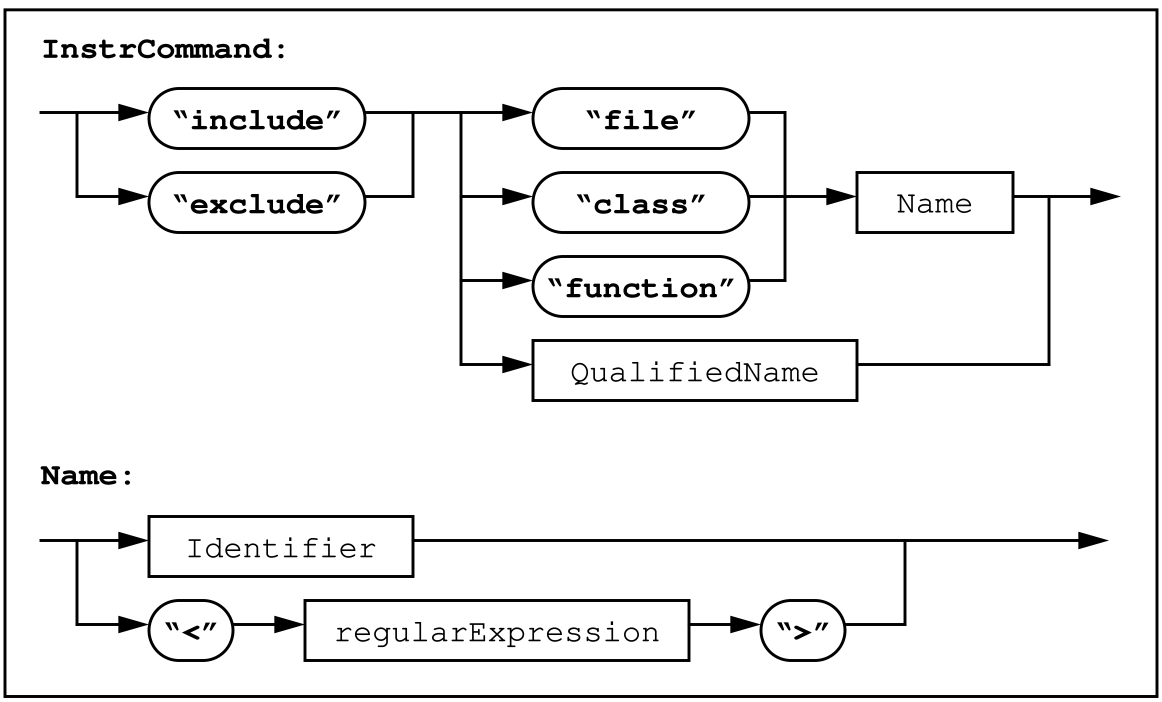

FIGURE 5. Syntax Diagram for "InstrCommand"

A command is either an include or an exclude command. The argument specifies what functions are instrumented (include) and which are not (exclude). The argument has basically two forms: Either it is “**`file`**”, “**`class`**”, or “**`function`**” followed by a name or it is a qualified name. “**`file`**” and “**`class`**” can be used to exclude/include all functions declared in one file or class. Name can be either a fixed identifier or a regular expression in angle brackets (see *regexp*(3)).

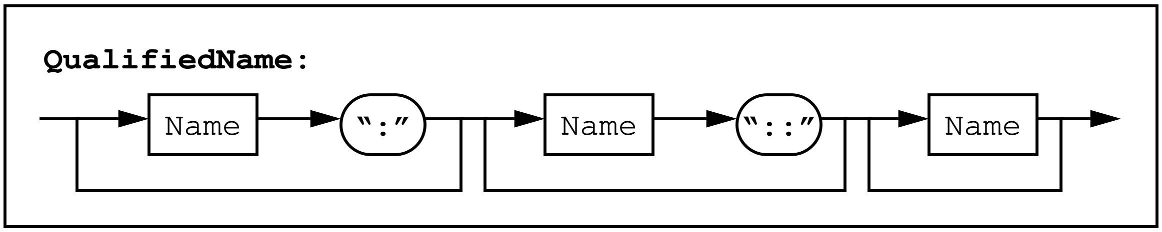

FIGURE 6. Syntax Diagram for “QualifiedName”

A qualified name can be used to specify complex combinations. It consists of a file name part followed by a colon, a class name part followed by a double colon, and a function name part. Each part of the qualified name can be empty. Also, like above, fixed identifiers or regular expressions in angle brackets can be used for each name part. Lets discuss some examples:

```
# exclude all functions in filenames ending with ".h"
exclude file <\.h$>

# exclude all member functions declared in classes
# Note that an empty regular expression matches always
exclude class <>

# exclude all functions which represent operators
# i.e., name doesn't start with a letter
exclude function <^[^A-Za-z]>

# include all functions of class "kernel"
# which contain "local" in their name
include kernel::<local>

# exclude all functions starting with a capitol letter
# declared in C++ files
exclude <(\.C|\.cc)$>:<^[A-Z]>
```

FIGURE 7. Examples of Instrumentation Commands

The order of the instrumentation commands in the file doesn’t matter. First of all, the instrumentor determines the *start set* of functions to instrument by processing all include commands. If there are no include commands, all functions in the source files are included. Next, the exclude commands are evaluated against the start set. The remaining functions are finally instrumented using the profiling approach described above.

3.3 Instrumentor Command Line Switches

The instrumentor is implemented as a stand-alone program and can be invoked by typing "`instr`" followed by the name of a .dep file. The .dep files can be generated by invoking "`pc++ -deponly` *`sourcefiles`*". The instrumentor recognizes the following options, which can be used to alter the default behavior:

`-h` or `-help` Print help information (long version of usage information including explanation)

`-u` or `-usage` Print usage information

`-v` or `-verbose` Turn on verbose mode

`-debug` Print useful debugging information

`-p` or `-profile` Insert special code for direct profiling instead of instrumenting for trace-based profiling

(`-c` or `-commands`) *`comfile`* Read instrumentation command file *comfile*. This option can be used repeatedly to specify more than one command file. It is recommended to use "`.prf`" as suffix for instrumentation command files.

(`-t` or `-tdl`) *`tdlfile`* Generate TDL description file *tdlfile*. It is recommended to use the suffix "`.tdl`" for TDL description files.

(`-e` or `-edf`) *`edffile`* Generate event description file *edffile*. It is recommended to use the suffix "`.edf`" for event description files.

(`-o` or `-output`) *`outfile`* Output the modified (instrumented) syntax tree into the .dep file named *outfile*. By default, instr overwrites the original .dep input file.

The resulting modified .dep file can be unparsed to C++ using "`dep2C++`" or "`unparse`". In general, the instrumentor will be called automatically from the pC++ compiler (see next subsection).

3.4 Integration into the pC++ Environment

This subsection describes the commands and tools which can be used to run direct profiling or trace-based profiling versions of pC++ programs.

3.4.1 pc++

If the pC++ compiler `pc++` is called with the `-profile` or `-trace` option, it automatically calls the instrumentor with the right flags to do all the necessary direct or trace-based function profiling instrumentation. Pc++ uses a system and a user instrumentation command file. The system instrumentation command file is located at "`$(SAGEROOT)/target/include/sysinstr.prf`". It may be empty. If there is a file with the same filename as the pC++ source program, but with the suffix "`.prf`" instead of "`.pc`", it is assumed that it is a user level instrumentation command file and also passed to the instrumentor.

Also, pc++ includes the necessary header files ("`profiler.h`" for profiling or "`se_event.h`" and "`tracer.h`" for tracing) automatically in the pC++ source program. Finally, it invokes the instrumentor which performs all necessary instrumentations and also generates a function symbol file or a TDL and EDF description file respectively. The function symbol file is named "`profile.ftab`". The TDL and EDF files will have the same basename as the pC++ input source file but with the suffix "`.tdl`" and "`.edf`" respectively.

3.4.2 MAKE

The *MAKE* utility provides an user-friendly and convenient way of compiling pC++ programs. It can be used by typing `MAKE <MachineName>`. If the *MachineName* is prefixed with "`prf-`" or "`trc-`" profiling or tracing versions of the program are generated respectively. The following *MachineNames* are supported:

- **Profiling**: uniproc, symmetry, tc2000, ksr, paragon, cm5
- **Tracing**: uniproc, symmetry, tc2000, ksr

MAKE calls first *pc++* with the necessary options, then uses the target's machine C++ compiler to generate the executable. There are also instrumented versions of the pC++ Kernel files and runtime system, both for direct and trace-based profiling. In addition to the instrumentation of user level functions, they provide profiling of runtime system functions[1] and collection access. In using *MAKE*, these special kernels are used in the linking phase.

3.4.3 pprof

Pprof is the parallel profile printing tool for pC++ profile datafiles.The output of *pprof* is similar to the normal UNIX *prof* tool. In addition, it prints a function profile for each node and also some collection data access statistics. Also, it prints a function profile summary (mean and sum) and collection data access summary for the whole parallel execution. The columns of the function profile table have the following meaning:

`%time`	The percentage of the total running time of the main program (Processor_Main) used by this function (on the specified node).
`msec`	The number of milliseconds accounted for by this function alone.
`total msec`	A running sum of the number of milliseconds accounted for by this function and all its childs (functions which are called within the current function).
`#calls`	The number of times this function was invoked.
`usec/call`	The average number of microseconds spent in this function per call.
`name`	The name of the function.

[1]. At the moment, only the Barrier runtime function is profiled. Future versions will include more routines.

Figure 8 shows the output of pprof of the UNIX uniprocessor version of the pC++ Simple benchmark:

```
NODE 0:
-------------------------------------------------------
            total        usec/
%time  msec  msec #call   call  name
-------------------------------------------------------
100.0    57   186     1 186694  Processor_Main
 69.0    55   128    64   2014  MyCollection::phello
 39.3    73    73    64   1148  MyElement::hello
  0.0     0     0     3     18  Barrier
  0.0     0     0     1     16  MyCollection::MyColl.

   local     remote   collection
accesses  accesses  num   name
     320         0    0   X

COLLECTION SUMMARY:
-------------------------------------------------------
MyCollection<MyElement> X, collection #0
        64 elements of size 16, 1-dimensional
        320 local / 0 remote accesses
```

FIGURE 8. Examples of pprof output

If a pC++ program compiled with the profiling option is executed, there will be a profile data file for each node the program was executed on. The profile data files will have names of the form `profile.<NodeId>`. If *pprof* is called without a parameter, it will look for all profile data files in the current directory and print profiles for all of them. But it is also possible to restrict the printing by specifying a list of node identifications. In addition to the normal parameters, *pprof* supports the following command line options:

`-c`|`-m`|`-t`	The mutually exclusive options c, m, and t determine the type of sorting of the output lines of the function profile. Sort by decreasing number of calls (-c), by decreasing time in milliseconds (-m), or by decreasing percentage of total time (-t) is supported. Sort by total time is the default.
`-r`	Reverse the order of sort to get reverse (ascending) order or smallest value first.
`-s`	Print only the summary profile information.
`-n` *num*	Print only the first *num* functions.

3.4.4 Trace Analysis

Currently, tracing of pC++ programs is restricted to pC++ ports to shared-memory parallel computer systems (Sequent Symmetry, BBN Butterfly, Kendall Square KSR-1) and the uniprocessor UNIX version. The port of the event tracing package to distributed memory machines is under way. Also, the pC++ programming environment does not yet contain any performance analysis or visualization tools. However, there are already some utility tools available that allow to analyze the resulting event traces with the help of externally available event trace analysis tools. Trace analysis involves several steps:

(1) **Merging**: If a pC++ program compiled with the tracing option is executed, there will be an event trace for each node the program was executed on. The trace files will have names of the form `<MachineName>.<NodeId>.trc`. These single node traces have to be merged into one global event trace, i.e., all event records have to be ordered according to increasing timestamps. This can be done with the tool *se_merge*. If the target machine does not have a hardware global clock, *se_merge* will also establish a global timescale for the event traces by correcting timestamps if necessary.

(2) **Trace Conversion, Analysis, and Visualization**: With the help of the utility tool *se_convert* and the EDF file, the traces can be converted to SDDF format to be used with the *Pablo* performance analysis environment [ROA+91, ROM+92] or to ALOG format to be used in the *Upshot* event display tool [HL91]. It also can produce a simple, user-readable ASCII dump of the binary trace.

The trace files can also processed with the SIMPLE event trace analysis environment or other TDL/POET based performance analysis tools [mohr90, mohr93a]. These tools use the generated TDL description of the trace files to access them.

Future versions of the pC++ parallel programming environment will have their own sophisticated trace analysis and visualization tools.

References

[BBG+92a] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Srinivas, "*Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools*", Technical Report, CICA, Indiana University, 1992.

[BBG+92b] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Yang, "*Distributed pC++: Basic Ideas for an Object Parallel Language*", Technical Report, CICA, Indiana University, 1992.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, "*Implementing a Parallel C++ Runtime System for Scalable Parallel Systems*", to appear in: Proceedings of the Supercomputing '93 Conference, Portland, Oregon, November 15-19, 1993.

[gann93] D. Gannon, "Libraries and Tools for Object Parallel Programming", in: *Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, Saint Hilaire du Touvet, France, Elsevier Science Publisher, Advances in Parallel Computing, Vol. 6, pp. 231-246, 1993.

[GKK82] S.L. Graham, P.B. Kessler, M.K. McKusick, "gprof: a call graph execution profiler", in: *Proc. SIGPLAN'82 Symp. on Compiler Construction*, 1982, pp. 120-126.

[HL91] V. Herrarte, E. Lusk, "*Studying Parallel Program Behavior with Upshot*", Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[mohr90] B. Mohr, "Performance Evaluation of Parallel Programs in Parallel and Distributed Systems", in: H. Burkhart (Eds.), *Proc. CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing*, Zürich, Lecture Notes in Computer Science 457, pp. 176-187, Berlin, Heidelberg, New York, London, Paris, Tokio, Springer Verlag, 1990.

[mohr93a] B. Mohr, "Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?", in: *Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing*, Saint Hilaire du Touvet, France, Elsevier Science Publisher, Advances in Parallel Computing, Vol. 6, pp. 231-246, 1993.

[mohr93b] B. Mohr, "*A Portable Software Event Tracing Module for Parallel Machines*", not yet written.

[ROA+91] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, B. A. A. Nazief, B. K. Totty, *Scalable Performance Environments for Parallel Systems*. In: Proceedings of the 6th Distributed Memory Computing Conference, IEEE, Computer Society Press, pp. 562-569, 1991.

[RAM+92] D. A. Reed, R. A. Aydt, T. M. Madhyastha, Roger J. Noe, Keith A. Shields, B. W. Schwartz, *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, November 1992.