Pablo® Instrumentation Environment User's Guide

Roger J. Noe*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

April 1, 1994

Copyright © 1992-1994 The University of Illinois Board of Trustees.
All Rights Reserved.

*Supported in part by the Defense Advanced Research Projects Agency under DARPA Contract No. DABT63-91-K-0004, by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by an equipment grant from the Digital Equipment Corporation External Research Program, and by a collaborative research agreement with the Intel Supercomputer Systems Division.

Contents

Introduction iii

1 Installing the Trace Capture Library 1
1.1 Trace Capture Library Compilation . . . 1
1.2 Using and Porting the Trace Capture Library . . . 1

2 Trace Capture Library Basic Features 3
2.1 Events and Event Identifiers . . . 3
2.2 Trace Record Types . . . 3
2.2.1 Full Trace Records . . . 3
2.2.2 Count Trace Records . . . 3
2.2.3 Interval Trace Records . . . 4
2.3 Elementary Instrumentation Calls . . . 4
2.3.1 traceEvent . . . 6
2.3.2 countEvent . . . 6
2.3.3 startTimeEvent and endTimeEvent . . . 6
2.3.4 endTracing . . . 6
2.4 Trace Output File Names . . . 6

3 Trace Capture Library Advanced Features 8
3.1 Event Configuration . . . 8
3.1.1 Trace Levels . . . 8
3.1.2 Event Rates . . . 9
3.1.3 Trace Functions . . . 9
3.1.4 Record Functions . . . 12
3.1.5 Miscellaneous Routines . . . 13
3.2 Event Defaults . . . 14
3.2.1 Default Trace Level . . . 14
3.2.2 Default Event Rate Limits . . . 14
3.2.3 Default Trace Function . . . 16
3.2.4 Default Record Function . . . 16
3.3 Defining New Trace Record Types . . . 16
3.4 Internal Events . . . 17
3.5 Additional Features . . . 17
3.5.1 Synchronization Control . . . 17
3.5.2 Buffer Size Control . . . 18
3.5.3 Socket Output . . . 18

4 Trace Capture Library Extensions 20
4.1 Procedure Tracing . . . 20
4.1.1 Procedure Trace Records . . . 20
4.1.2 Procedure Tracing Initialization . . . 25
4.2 Message Tracing . . . 26
4.2.1 Message Passing Trace Records . . . 26

4.2.2 Message Tracing Initialization 32
4.2.3 Intel iPSC Message Tracing 32
4.2.4 Thinking Machines CMMD Message Tracing 32
4.3 Loop Tracing 33
4.3.1 Loop Trace Records 33
4.3.2 Loop Tracing Initialization 33
4.4 Input/Output Tracing 34
4.4.1 Basic Input/Output Trace Records 34
4.4.2 Summary Input/Output Trace Records 43
4.4.3 Intel Paragon Input/Output Tracing 44
4.4.4 Input/Output Tracing Initialization 48

5 Preparing Trace Files for Visualization **49**
5.1 Merging Trace Files 49
5.2 Conversion of Trace Files 49

A A Representative Example **51**
A.1 Application Makefile 51
A.2 Host Application Code 51
A.3 Node Application Code 51
A.3.1 Instrumentation Initialization 51
A.3.2 Run-Time Instrumentation 52
A.3.3 Trace Data Collection 52

List of Figures

1 demo.c Before Instrumentation 4
2 demo.c After Instrumentation 5
3 Setting the Trace Level 8
4 Setting the Event Rate Limits 10
5 Adaptive Tracing Source Code 11
6 Setting the Event Trace Function 11
7 Event Record Function (declaration) 12
8 Setting the Event Record Function 13
9 Multiple Event Parameter Configuration 13
10 Resetting the Event Count 14
11 Canceling an Event Interval Timer 14
12 Setting the Default Trace Level 15
13 Setting the Default Event Rate Limits 15
14 Setting the Default Trace Function 16
15 Setting the Default Record Function 16
16 Setting the User Record Descriptor Function 17
17 Procedure Entry Trace Record Descriptor (ASCII) 21
18 Procedure Exit Trace Record Descriptor (ASCII) 22
19 Procedure Entry Count Record Descriptor (ASCII) 23
20 Procedure Exit Count Record Descriptor (ASCII) 24
21 Blocking Message Send Begin Record Descriptor (ASCII) 27
22 Blocking Message Send End Record Descriptor (ASCII) 27
23 Blocking Message Receive Begin Record Descriptor (ASCII) 28
24 Blocking Message Receive End Record Descriptor (ASCII) 28
25 Nonblocking Message Send Begin Record Descriptor (ASCII) 29
26 Nonblocking Message Send End Record Descriptor (ASCII) 29
27 Nonblocking Message Receive Begin Record Descriptor (ASCII) 30
28 Nonblocking Message Receive End Record Descriptor (ASCII) 30
29 Miscellaneous Message Global Operation Begin Record Descriptor (ASCII) 31
30 Miscellaneous Message Global Operation End Record Descriptor (ASCII) 31
31 Loop Entry Trace Record Descriptor (ASCII) 34
32 Loop Exit Trace Record Descriptor (ASCII) 35
33 Loop Entry Count Record Descriptor (ASCII) 36
34 Loop Exit Count Record Descriptor (ASCII) 37
35 Input/Output Read Record Descriptor (ASCII) 38
36 Input/Output Write Record Descriptor (ASCII) 39
37 Input/Output Seek Record Descriptor (ASCII) 40
38 Input/Output Open Record Descriptor (ASCII) 41
39 Input/Output Fopen Record Descriptor (ASCII) 42
40 Input/Output Close Record Descriptor (ASCII) 43
41 Input/Output File Lifetime Summary Record Descriptor (ASCII) 45
42 Input/Output Time Summary Record Descriptor (ASCII) 46
43 Input/Output Block Summary Record Descriptor (ASCII) 47

44 Input/Output Histogram Summary Record Descriptor (ASCII) 48

Introduction

This user guide is intended as an introduction to using the Pablo[1] Instrumentation Environment. The intended audience includes all users of the Pablo instrumentation library and the iPablo instrumenting parser interface.

The papers *The Pablo Performance Analysis Environment*, *An Informal Guide to Using Pablo*, and *The Pablo Self-Defining Data Format* are helpful in placing the instrumentation environment in the context of the entire Pablo system. The reader should refer to these documents for background information and definition of unfamiliar terminology.

The software discussed in this guide is contained within the Pablo *trace capture library*, sometimes called the instrumentation library or just the trace library. The purpose of this library is to provide a coherent set of basic routines that will capture trace information for arbitrary parallel applications. This may be achieved by having the user insert calls to library routines into the application source code (both C and Fortran language interfaces are provided). The user then executes the application, producing trace data files which may be analyzed using the Pablo Visualization Environment. Alternatively, the user may employ an instrumenting parser, which permits the user to specify instrumentation points in a C language source code file via a graphical interface. The parser then inserts the requested instrumentation library calls into the source code and supplies necessary library initialization source code. Use of this instrumenting parser and its graphical interface is described in the *iPablo User's Guide*.

Section 1 of this guide provides guidelines to successfully installing the Pablo Instrumentation Environment software onto a target system, making it available for use. The basic functionality of the trace capture library is presented in section 2, assuming no special programmer expertise on the part of the reader. Section 3 gets more esoteric, introducing less frequently used features of the instrumentation environment. Some programming ability and access to the trace library source code is assumed here. This section may be skipped by the casual user. Some extensions to the trace library itself are discussed at length in section 4. This section serves two purposes, affording the user interested in making quick gains with the instrumentation environment an opportunity to apply commonly desired tracing functionality to specific applications, while simultaneously giving the sophisticated user with significant programming experience a tutorial in designing custom extensions to the Pablo Instrumentation Environment. Section 5 explains the use of a few tools needed to prepare trace data output files for analysis, serving as the bridge between the Instrumentation Environment and the Pablo Visualization Environment. Finally, appendix A "annotates" an example application, assuming the user has access to its source code, which is provided as part of the Pablo distribution.

[1] "Pablo" is a registered trademark of the Board of Trustees of the University of Illinois, registered in the U.S. Patent and Trademark Office.

1 Installing the Trace Capture Library

1.1 Trace Capture Library Compilation

It is first necessary to compile the trace capture library for whatever system its use is intended. This distribution of the trace capture library source code has been ported to the Intel iPSC/2 and iPSC/860 systems, the Intel Paragon XP/S, and to the Thinking Machines CM-5 using CMMD. Also provided is a sequential single-processor version which should be portable to most BSD UNIX implementations (version 4.1 and later).

The appropriate version of the trace library software is compiled by making the top directory of the Pablo Instrumentation Environment source distribution the current directory (this should have a name ending with `Instrument`), editing the file `Makefile` found within that directory, and executing the command `make all`. This will compile the trace capture library source code and extension software, depositing the object files into archive library files named `libPabloTrace.a` and `libPabloTraceExt.a`. These archive files may then be linked with applications. For ease of use, these archive libraries may be installed along with necessary header files by specifying installation directory path names in the top-level instrumentation `Makefile` (see especially the `LIBDIR` and `INCDIR` macros within) and executing the command `make install`.

It's possible that this compilation will not succeed on the first attempt. Successful compilation of the trace capture library is dependent on the local system configuration. For example, the Intel system version of the trace library was developed on a machine with a C cross-compiler for the hypercube, named **icc**. Since not every system will use this same name, it may be necessary to change references to this compiler to the correct local target compiler's name. Watching the `make` output will be a good clue as to where (i.e. in which directory) the problem occurred. Most of these problems can be resolved or even avoided by carefully reading the top-level `Makefile` and following the directions for local customization contained within the comments of that file.

Alternatively, the user may edit the file `Makefile.defines` found in the top-level directory of the Pablo distribution, the parent of the above-mentioned `Instrument` directory. A `gnumake` performed at this level will compile the entire Pablo visualization and instrumentation environments. The user should be aware that compilation of the trace library will fail if the appropriate compilation tools for the target system are not available on the host system (the one on which the compilation is being done). Depending on the local file system configuration, it may be necessary to split this larger `make` job into smaller tasks. For example, the trace library alone could be compiled on the target system itself, then all objects could be copied back onto a workstation to complete the job.

1.2 Using and Porting the Trace Capture Library

The `Makefile`s under the `Examples` subdirectories may be used as patterns for compiling user applications with the trace capture library. The `libPabloTrace.a` file may of course be copied to any place the user finds it convenient to install, or automatically installed by setting the `LIBDIR` macro in the top-level `Makefile` to the full path name of the desired directory, and then executing `make install`. This is discussed in greater detail in section 4 and appendix A.

It is possible to modify the system-dependent portion of the trace capture library code so that it may be used on systems other than that for which it has currently been implemented. Further discussion of this is well beyond the scope of this guide, although not beyond the abilities of a sophisticated user who wishes to delve into the system-dependent portions of the trace capture

library code and find analogous system calls and data structures for the new target system.

2 Trace Capture Library Basic Features

This section discusses elementary trace capture concepts of events, event identifiers, and trace record types, and introduces the few trace-generating library routines needed to do the most basic instrumentation of an application.

2.1 Events and Event Identifiers

In the scope of this guide, an *event* is an instance of the execution of a specific statement or instruction in an application, on a specific processor. This event may be any type of execution-time occurrence chosen by the user, subject to source code language limitations. Common examples of this are entrance to or exit from a specific procedure, entrance to or exit from a specific loop, or invocation of a system call, such as a message-passing routine. The Pablo Instrumentation Environment permits the user to assign distinct *event identifiers* to each of the defined events, using integers as these identifiers. Choice of these event identifiers, or event IDs, is left completely up to the user, except for several *reserved* event IDs used for events internal to the trace capture library.[2] It is this event ID that appears in the captured trace data which permits the identification of a trace record with a specific location in the instrumented application code.

2.2 Trace Record Types

In addition to an event ID, each captured trace record will contain other data characterizing a specific event. These include a timestamp, indicating the number of machine-specific clock ticks elapsed since the trace capture library was initialized when the event occurred, and a number indicating on which processor (or node) the event was recorded. Each event type may also generate event-specific data which is included within the trace records of events bearing that event ID. For any given event ID, there may be as many as three different types of trace records generated for events corresponding to that event ID. These are enumerated in the following paragraphs.

2.2.1 Full Trace Records

The simplest type of trace record is one which indicates only that the specified event has occurred. This concept is extended in the Pablo Instrumentation Environment by enabling the user to append an arbitrary amount of data, defined both in format and content by the user, to the resulting trace record. This is called a *full* trace record. When the user chooses to append no additional data, it is called a *simple* trace record.

2.2.2 Count Trace Records

The trace capture library keeps track of the number of times each event ID occurs during execution. The *count* type of trace record differs from the simple trace record by the inclusion of this running count within the captured record.

[2] These reserved internal event IDs are normally coded to be large magnitude negative integers. They may be reassigned by changing the relevant lines in the trace library source file `TraceParam.h` found in the subdirectory `TraceLibrary/Portable/Includes` and then recompiling the entire library.

```
main() {
        int     i;

        for ( i = 1; i <= 10; i++ ) {
                testFunction( i );
        }
}

testFunction( n )
int     n;
{
        int     j;

        for ( j = 0; j < n; j++ ) {
                /* Arbitrary non-branching code goes here */
        }
}
```

Figure 1: demo.c Before Instrumentation

2.2.3 Interval Trace Records

The user may also choose to use an event ID to correspond to the amount of time which elapses between two specific points in the instrumented application. By starting an interval timer (implemented within the trace capture library) at one point, it may be later stopped at another point. When a running interval timer is stopped, an *interval* type of trace record is generated. This type of record differs from the simple trace record by the inclusion of the time interval amount, measured in the same units as the timestamp element of the trace output records.

2.3 Elementary Instrumentation Calls

Each of these general types of trace records may be generated by inserting calls to the appropriate routines provided by the trace capture library. Consider the fragment of source code in figure 1.[3] Perhaps the user of this rather dubious bit of code desires to trace events corresponding to the top and bottom of the `for` loop in the function `main`, entrances to and exits from the procedure `testFunction`, the beginning of the unspecified code within the `for` loop of that procedure, and have an interval timer show the total execution time. In this case, the user might add calls to procedures provided by the trace capture library to arrive at the source code in figure 2. The effects of these instrumentation calls are explained in the following subsections.

[3] All example source code in this guide will be in the C language; the Fortran interface routines are identical except for upper/lower case differences.

```
main() {
        int     i;

        startTimeEvent( 0 );
        for ( i = 1; i <= 10; i++ ) {
                traceEvent( 1, "top of for(i) loop", 18 );
                testFunction( i );
                countEvent( 2 );
        }
        endTimeEvent( 0 );
        endTracing();
}

testFunction( n )
int     n;
{
        int     j;

        traceEvent( 3, "entered testFunction", 20 );
        for ( j = 0; j < n; j++ ) {
                traceEvent( 2, "inside for(j) loop", 18 );
                /* Arbitrary non-branching code goes here */
        }
        traceEvent( 4, "exiting testFunction", 20 );
}
```

Figure 2: demo.c After Instrumentation

2.3.1 traceEvent

The trace library routine **traceEvent** generates full trace records. The arguments to it are:

- The event identifier (an integer). Here the event ID 1 indicates the top of the `for` loop in `main`, 2 indicates the top of the `for` loop in `testFunction`, 3 indicates the entrance of `testFunction`, and 4 indicates its exit.
- A pointer to user-defined additional data, or a null pointer in the case of simple trace records. In this example, all the pointers are to character strings, but may be to any address accessible to the application.
- The size (in bytes) of the user-defined data, zero for a simple trace record.

2.3.2 countEvent

The trace library routine **countEvent** generates count trace records. The only argument to it is an event ID. Only one event ID is used in this example to generate count trace records, event ID 2, at the bottom of the `for` loop in `main`. Note that this event ID is being used to generate both full trace records and count type trace records. In this example this practice has questionable utility, but this is permitted by the trace capture library.

2.3.3 startTimeEvent and endTimeEvent

The trace library routines **startTimeEvent** and **endTimeEvent** are used to operate an event interval timer. Both take just one argument, an event ID. When **endTimeEvent** is executed for an event ID with a matching previous call to **startTimeEvent**, the running interval timer is stopped and an interval trace record is generated. Event ID 0 is therefore used in this example to produce an interval event with the interval time in the resulting trace record indicating the execution time of the program. Calls to **startTimeEvent** and **endTimeEvent** may be nested, so long as they are paired properly. Each invocation of **startTimeEvent** starts a new interval timer for the argument event ID. Calls to **endTimeEvent** stop only the most recently started interval timer for the argument event ID.

2.3.4 endTracing

During execution of an application with the trace capture library, trace records are buffered in memory until it is appropriate to dump them. (Specific criteria for what is meant here by "appropriate" are discussed in more detail in *The Pablo Performance Analysis Environment.*) At the conclusion of a trace capture phase of execution, the trace library routine **endTracing** must be called to flush whatever data may remain in the trace buffer to the trace output file. In particular, **endTracing** must always be called before the application terminates to prevent loss of data.

2.4 Trace Output File Names

The preceding sections present all the information that is absolutely necessary to using the trace capture library effectively for simple applications. However, it may be helpful at this point to digress a bit and point out that the trace library provides the user with a simple way to change the name of

the trace output file. The trace library routine **setTraceFileName**, when passed a character string as an argument, will use that string as the name of the trace output file to use. This routine must be called before the trace library is initialized, i.e. before any of the above interface routines have been called. (But when **endTracing** is called, the library uninitializes, so the trace file name may be changed for different trace capture phases of execution in one instrumented application.) The user may wish to have the node or processor number included in the trace file name in some way. Section 5 discusses how to merge several trace output files into one output file for an application.

```
int     eventID;
int     traceLevel;

/* Must have LEVEL_IGNORE <= traceLevel <= LEVEL_MAXIMUM */

if ( setEventLevel( eventID, traceLevel ) == SUCCESS ) {

        /* trace level has been changed for eventID */

}
```

Figure 3: Setting the Trace Level

3 Trace Capture Library Advanced Features

In addition to the simple interface routines presented above in section 2, the trace capture library provides a number of routines which permit the user to customize the behavior of the trace library to the application. Chief among these advanced interface routines are those which modify parameters associated with an event ID.

3.1 Event Configuration

Associated with each event ID is a set of parameters, some of which may be modified by the instrumented application at run time. The following subsections discuss these parameters and associated configuration routines.

3.1.1 Trace Levels

Each event ID has associated with it at every point in time a current *trace level*. This may be thought of as something like a priority level, with higher trace levels indicating greater importance of a specific event. Trace levels for an event ID may vary during execution of an instrumented application, from a minimum of `LEVEL_IGNORE` to a maximum of `LEVEL_MAXIMUM`, both constants defined in the source code file `TraceLibrary/Portable/Includes/TraceParam.h` and compiled into the trace capture library code. In between are defined three threshold trace level values, `LEVEL_FULL` for full trace events, `LEVEL_COUNT` for count events, and `LEVEL_INTERVAL` for interval events, all defined in the `TraceParam.h` file.[4] For a trace record to be generated for an event, the event ID's current trace level must be no less than the corresponding threshold, depending on which interface routine (**traceEvent**, **countEvent**, or **endTimeEvent**) was invoked for the event.

The maximum trace level for an event ID may be set by invoking the interface function **setEventLevel**, passing the event ID and the desired trace level as arguments (see figure 3). Once this maximum is set, the event ID's current trace level will not be permitted by the trace library to

[4] All the symbolic constants referred to in this guide are defined in the `TraceParam.h` file and compiled into the trace capture library. They may be changed by the user prior to trace library compilation and installation, subject to the comments found within that file.

exceed the configured maximum. Notably, if an event ID's trace level is set to `LEVEL_IGNORE` then the trace library will do nothing with that event; it is ignored. If an event ID's trace level is set to some legal value greater than `LEVEL_IGNORE` then the trace library will not decrease that event ID's trace level to the point it is ignored (but see the subsection on Trace Functions below for related information). Setting the maximum trace level for an event ID also sets its current trace level to the same value.

3.1.2 Event Rates

Changing an event ID's trace level is typically a one-time occurrence in an instrumented application. The trace capture library provides a mechanism to automatically and dynamically vary an event ID's current trace level, between the minimum value just above `LEVEL_IGNORE` and the maximum value passed to **setEventLevel**. This mechanism is used to substitute event count trace records for full trace records and interval trace records (independently) when the particular event ID is occurring too rapidly. This technique is called event *throttling* and the specific algorithm employed by the trace capture library is known as *adaptive tracing.*

In the adaptive tracing algorithm, two parameters are associated with each event ID, called the low and high event latency limits. Both parameters are measures of time, using the same units as for the timestamp fields in the trace output data. The low latency limit is the minimum amount of time which may pass between successive events of the particular event ID without throttling down on that event ID. That is, if the incremental event rate for this event ID is so high that two such events occur closer together in time than that measured by the low latency limit, then its current trace level will be decreased by one (but not down to `LEVEL_IGNORE`). As might be expected, the high latency limit is the maximum amount of time which may pass between successive events of the particular event ID without throttling up on that event ID. If the incremental event rate for the event ID is so low that two such events occur farther apart than that measured by the high latency limit, then its current trace level will be increased by one (but not above the configured maximum).

By default, the low latency limit is set so low and the high latency limit so high that adaptive tracing is effectively disabled. This may be changed by invoking the trace capture library interface function **setEventRateLimits**, passing the event ID, low and high latency limits as arguments (see figure 4). The low latency limit must be nonnegative and must not exceed the high latency limit.[5] It is important to note that adaptive tracing is *deactivated* within the trace capture library by default and, like many other parameters associated with each event ID, may be overridden by the user. This is the topic of the following subsection.

3.1.3 Trace Functions

When an application indicates to the trace capture library that an event has occurred (by invoking one of the simple interface routines discussed in section 2) the trace library must determine whether or not a trace record is to be generated for the event. This is achieved by invoking the *trace function* associated with the particular event ID. One such example is the adaptive tracing algorithm referred to above. The source code for this function is shown in figure 5, with comments removed for brevity.

[5] The type for latency limits is defined in the include file `Trace.h`. The members of the `CLOCK` structure type denote the low- and high-order halves of a 64-bit integer value. This same type is used to represent timestamps and other integer representations of time values.

```
int     eventID;
CLOCK   lowLatency;
CLOCK   highLatency;

/* Must have 0 <= lowLatency < highLatency */

if ( setEventRateLimits( eventID, lowLatency, highLatency ) == SUCCESS ) {

        /* event rate limits have been changed for eventID */

}
```

Figure 4: Setting the Event Rate Limits

(This function is defined in the trace library source file `TraceLibrary/Portable/Internal.c`.) The first argument to the trace function is a pointer to the event ID's *event descriptor*, the structure of which is defined in a `typedef` statement in the file `TraceLibrary/Portable/Includes/Trace.h`. The event descriptor contains all the parameters and trace library internal data structures associated with the event ID. The second argument passed to the trace function is the current time, in the form of a `CLOCK` type structure, also defined in `Trace.h`. The trace function may examine any elements of the event descriptor and may even change some elements, such as the current ("soft") trace level `eventSoft` and any of the parameters listed in this section as being user-configurable. The trace function *must* return a legal trace level as defined by the symbolic constants in the `TraceParam.h` file.

The trace capture library interprets the return value of the event ID's trace function as being the current trace level of the specific event. If this is `LEVEL_IGNORE` then the trace library does nothing further with this event. Otherwise, the trace library increments the event count and records the current time in the event ID's event descriptor. Whether or not a trace record is then generated for this event depends on the aspect of the current trace level to the relevant threshold value.

Any user-supplied function which satisfies the above trace function requirements (same argument types accepted, legal trace level returned) may be substituted as the trace function for an event ID in place of the adaptive tracing routine. This is accomplished by passing a pointer to the desired function as an argument to the trace capture library interface routine **setEventTraceFunction** as shown in figure 6. Passing a null function pointer to **setEventTraceFunction** will cause the trace library not to invoke any trace function for the given event ID, thus its current trace level does not change. This is the default for all event IDs, for reasons of improved performance. To enable the adaptive tracing algorithm for a specific event ID, pass the `adaptiveTrace` function pointer (declared in the `Trace.h` include file) to **setEventTraceFunction**.

This capability gives the user great power and flexibility in defining application-specific data structures, implementing interaction between event IDs (seemingly separate, to the trace library) and an almost unlimited number of other possibilities. Some of these possibilities are exploited to great advantage in the software discussed in section 4.

```
int
adaptiveTrace( eventPointer, currentTime )

TR_EVENT        *eventPointer;
CLOCK           currentTime;
{
        CLOCK           timeDelta;

        if ( eventPointer->eventHard == LEVEL_IGNORE )
                return LEVEL_IGNORE;

        if ( clockCompare( eventPointer->eventLast, noSuchClock ) != 0 ) {
                timeDelta =
                        clockSubtract( currentTime, eventPointer->eventLast );

                if (( eventPointer->eventSoft < eventPointer->eventHard ) &&
                    ( clockCompare( timeDelta, eventPointer->eventHighLatency )
                    > 0 ))
                        eventPointer->eventSoft++;

                if (( eventPointer->eventSoft > (LEVEL_IGNORE + 1) ) &&
                    ( clockCompare( timeDelta, eventPointer->eventLowLatency )
                    < 0 ))
                        eventPointer->eventSoft--;
        }

        return eventPointer->eventSoft;
}
```

Figure 5: Adaptive Tracing Source Code

```
        int     eventID;
        int     (*traceFunction)();

        if ( setEventTraceFunction( eventID, traceFunction ) == SUCCESS ) {

                /* trace function has been changed for eventID */

        }
```

Figure 6: Setting the Event Trace Function

```
TR_RECORD *
externalEventRecord( recordType, eventPointer, timeStamp,
                     dataPointer, dataLength )

int             recordType;
TR_EVENT        *eventPointer;
CLOCK           timeStamp;
char            *dataPointer;
unsigned int    dataLength;
```

Figure 7: Event Record Function (declaration)

3.1.4 Record Functions

After the trace capture library has determined that a trace record ought to be generated for an event, it invokes the event ID's *record function* to perform the actual allocation and population of the fields of the resultant trace record. By default, external (as distinguished from internal) event IDs use a trace capture library internal routine called `externalEventRecord`, defined in the source code file `TraceLibrary/Portable/Internal.c`. The beginning of the function declaration for this routine is shown in figure 7. The arguments to a record function are:

- The record type, one of the symbolic constants `RECORD_TRACE` (for a trace record generated from a call to **traceEvent**), `RECORD_COUNT` (from a call to **countEvent**), or `RECORD_INTERVAL` (from a pair of calls to **startTimeEvent** and **endTimeEvent**).
- The pointer to the event ID's event descriptor.
- A timestamp to be used in place of the `eventLast` value stored in the event descriptor. (This argument exists for historical reasons of generality. In this software release, this argument always has the value `noSuchClock`, a symbolic constant, meaning the timestamp stored in the event descriptor is always to be used.)
- A pointer to user-defined data (a null pointer except for full traces).
- The number of bytes of user-defined data (zero except for full traces).

The method by which a record function creates and fills in a trace record with the correct data is complicated and requires thorough knowledge of the SDDF (self-defining data format) definition. This is probably best understood by examining source code (as in the `Internal.c` file), but see also section 4. For the purposes of the present discussion it suffices to note that the value returned by a record function is a valid pointer to a *record descriptor*, the structure of which is defined in a `typedef` statement in the source code file `TraceLibrary/Portable/Includes/Trace.h`.

A user-supplied function which satisfies the above record function requirements (same argument types accepted, legal record descriptor pointer returned) may be substituted as the record function for an event ID in place of the default routine. This is accomplished by passing a pointer to the desired function as an argument to the trace capture library interface routine **setEventRecordFunction** as shown in figure 8. The trace capture library also supplies a trivial record function

```
int             eventID;
TR_RECORD       *(*recordFunction)();

if ( setEventRecordFunction( eventID, recordFunction ) == SUCCESS ) {

        /* record function has been changed for eventID */

}
```

Figure 8: Setting the Event Record Function

```
int             eventID;
unsigned int    traceLevel;
CLOCK           lowLatency;
CLOCK           highLatency;
int             (*traceFunction)();
TR_RECORD       *(*recordFunction)();

if ( configureEvent( eventID, traceLevel, lowLatency, highLatency,
                     traceFunction, recordFunction ) == SUCCESS ) {

        /* parameter(s) has (have) been changed for eventID */

}
```

Figure 9: Multiple Event Parameter Configuration

called **nullRecordFunction** which always returns an empty (zero size) trace record, regardless of the arguments passed to it. This could be used by an application to prevent an event ID from generating any trace data without interfering with its trace function.

3.1.5 Miscellaneous Routines

- The trace capture library interface routine **configureEvent** is provided as a convenience to the user who wishes to configure more than one of the foregoing parameters for an event ID without having to invoke multiple interface routines. Its usage is shown in figure 9. If at least one of the configurable parameters passed to this routine meets the requirements stipulated in the relevant subsection above, then **configureEvent** will return `SUCCESS`, otherwise `FAILURE`. The user may pass illegal values for parameters which are not to be reconfigured.
- The trace capture library interface routine **resetCount** may be invoked to reset the count of the number of occurrences of an event ID to zero (figure 10).

```
int             eventID;

if ( resetCount( eventID ) == SUCCESS ) {

        /* count has been reset for eventID */

}
```

Figure 10: Resetting the Event Count

```
int             eventID;

if ( resetInterval( eventID ) == SUCCESS ) {

        /* interval timer has been canceled for eventID */

}
```

Figure 11: Canceling an Event Interval Timer

- The trace capture library interface routine **resetInterval** may be invoked to cancel the most recently started interval timer for an event ID (figure 11). For this to succeed there must be at least one interval timer for the given event ID running, i.e. there must have been a previous call to **startTimeEvent** for this event ID with no matching **endTimeEvent** yet invoked.

3.2 Event Defaults

Each of the configurable event parameters listed above has a corresponding default value. When an event ID is initialized by the first reference to it in a trace capture library interface routine, that event ID is configured with the default parameters in effect at the time. While these defaults have compiled-in initial values, their current values may be changed by invoking interface routines listed below. This permits large numbers of event IDs to be configured identically with great ease. Note that a call to **endTracing** will restore the current value of these defaults to their initial values.

3.2.1 Default Trace Level

Use of the trace capture library interface routine **setDefaultTraceLevel** is shown in figure 12.

3.2.2 Default Event Rate Limits

Use of the trace capture library interface routine **setDefaultRateLimits** is shown in figure 13.

```
int     traceLevel;

/* Must have LEVEL_IGNORE <= traceLevel <= LEVEL_MAXIMUM */

if ( setDefaultTraceLevel( traceLevel ) == SUCCESS ) {

        /* default trace level has been changed */

}
```

Figure 12: Setting the Default Trace Level

```
CLOCK   lowLatency;
CLOCK   highLatency;

/* Must have 0 <= lowLatency < highLatency */

if ( setDefaultRateLimits( lowLatency, highLatency ) == SUCCESS ) {

        /* default rate limits have been changed */

}
```

Figure 13: Setting the Default Event Rate Limits

```
int     (*traceFunction)();

if ( setDefaultTraceFunction( traceFunction ) == SUCCESS ) {

        /* default trace function has been changed */

}
```

Figure 14: Setting the Default Trace Function

```
TR_RECORD       *(*recordFunction)();

if ( setDefaultRecordFunction( recordFunction ) == SUCCESS ) {

        /* default record function has been changed */

}
```

Figure 15: Setting the Default Record Function

3.2.3 Default Trace Function

Use of the trace capture library interface routine **setDefaultTraceFunction** is shown in figure 14. Note that the adaptive tracing algorithm discussed above may be selected as the default by passing a pointer to `adaptiveTrace` as the argument to **setDefaultTraceFunction**.

3.2.4 Default Record Function

Use of the trace capture library interface routine **setDefaultRecordFunction** is shown in figure 15.

3.3 Defining New Trace Record Types

If an application needs to define a specialized type of trace record, it must first define a function which, when invoked, will generate the record descriptor(s) for that type of trace record. The method by which such a function creates and outputs these record descriptors is complicated and requires thorough knowledge of the SDDF (self-defining data format) definition. This is probably best understood by examining source code and is discussed in more detail in section 4. For the purposes of the present discussion it suffices to note that the trace capture library must be informed which user-supplied function will perform this task. This is accomplished by passing a pointer to the designated function to the trace library interface routine **setRecordDescriptor** (see figure 16). This call to **setRecordDescriptor** *must* be made before any trace records have been generated in the current trace capture phase of execution.

```
int     (*recordDescriptorFunction)();

if ( setRecordDescriptor( recordDescriptorFunction ) == SUCCESS ) {

        /* record descriptor function will be invoked */

}
```

Figure 16: Setting the User Record Descriptor Function

3.4 Internal Events

The trace capture library provides a set of internal events, primarily for the purpose of inserting additional performance data into the trace output. For example, whenever a trace buffer is dumped to the trace output file, an internal event with the reserved event ID `EVENTID_DUMPTIME` is generated, resulting in a trace record containing a field value indicating how much time the trace data dump required.

However, there is one internal event which exists not to generate trace records, but to exert some control over other events. An internal event with the reserved event ID `EVENTID_AGGREGATE` is generated by the trace library upon every occurrence of an external event. This happens immediately after the external event's tracing function returns a current trace level indicating that the external event ID is not being ignored. The purpose of this internal event is to enable the user to control the *aggregate event rate* by throttling all external event IDs simultaneously, even though their individual event rates might not warrant separate throttling. Like external events, this internal event uses the adaptive tracing algorithm to determine its current trace level. The user may enable this feature by invoking some of the event configuration interface routines presented above on the internal event ID.

If the aggregate event rate-controlling internal event's current trace level is less than the external event's current trace level (as returned by its trace function), then the internal event's current trace level will take precedence and its value will be used as the effective trace level of this event. This may affect the trace library's subsequent determination of whether or not the external event ID's record function is to be invoked, and with what record type argument.

3.5 Additional Features

The trace capture library provides other interface functions which may be invoked by an application to control other aspects of the trace library's behavior, aspects which are independent of specific events.

3.5.1 Synchronization Control

Ordinarily, the trace library accumulates trace records in an internal buffer and then periodically outputs the contents of this buffer. Except for the sequential single-processor portable UNIX version of the trace library, the trace library synchronizes all the nodes of a partition executing a given

application so that they all dump their trace buffers simultaneously. This behavior, synchronous trace library operation, is the default. An application may disable this behavior and force *asynchronous* trace library operation by invoking the interface function **setTraceLibraryMode** with the argument `MODE_ASYNC`, a symbolic constant defined in `TraceParam.h`. This will also disable the clock synchronization algorithms employed by the trace library on those parallel machines which do not make available a global clock. The **setTraceLibraryMode** function may only be invoked prior to trace library initialization.

3.5.2 Buffer Size Control

The size of the trace buffer used by the library is by default a compiled-time constant found in `TraceParam.h`. This may be changed by an application by invoking the interface function **setTraceBufferSize**. This function takes a single argument, a non-negative integer indicating the trace buffer size in bytes. An argument value of zero is expressly permitted, which will cause the trace library to operate in *unbuffered* mode. That is, each trace record will be output individually, with no accumulation of trace records in an internal buffer. By default, unbuffered mode implies asynchronous mode, as discussed in the previous paragraph. The trace buffer size may only be selected prior to trace library run-time initialization.

3.5.3 Socket Output

Typically, an application would invoke **setTraceFileName** with an argument containing the name of the trace file which should be generated by that node. Trace library defaults provide generic trace file names, such as `Pablo0000.bin`, with the numeric digits indicating the node number. An application may elect instead of generating trace files to send trace records directly to an Internet domain *socket*. This is accomplished by invoking the trace capture library interface function **setTraceSocketAddress** with three arguments:

- The Internet protocol address of a destination host on which a user-supplied server is listening.
- A valid datagram socket (UDP) port to which trace data records will be output.
- A valid stream socket (TCP) port to which the SDDF header and trace record descriptors will be output.

The server address argument is mandatory and must be in the long integer Internet address format, such as the return value of `inet_addr` or the `h_addr` member of a `hostent` structure returned by `gethostbyname`. The datagram port argument is also mandatory and must be a positive integer. The application must provide a server which accepts datagrams on that port at the given address if any trace data is to be received. Note that the various nodes executing an application may all send their trace data to the same datagram port, they may send trace data to separate ports, some nodes may output to trace files while others output to sockets, or any combination in between. The user *must* select a trace buffer size which is no larger than the maximum permitted datagram size. This varies from one implementation to another and is difficult to determine at run time, but a buffer size of 8K bytes or less will probably be a safe choice. Selecting too large a trace buffer size will result in loss of data. Finally, the stream port argument to **setTraceSocketAddress** is optional. If a positive integer is supplied, the trace library will output the SDDF header and trace

record descriptors to that stream (not datagram) socket destination when the trace library becomes initialized. This socket will then be held open in a connected state until after the last trace data record has been written to the datagram socket, at which time the stream socket will be closed. If a negative number is passed for the stream port, no TCP socket will be opened by the trace library on this node and the trace record descriptors will be discarded.

The socket output address may only be selected prior to trace library initialization. A subsequent invocation of **setTraceFileName** will cancel the selection of socket output and *vice versa*; the options of trace file output and socket output are mutually exclusive.

4 Trace Capture Library Extensions

Given the flexibility of the user's interface with the trace capture library and the generality of the library's design, it is not difficult for a user to extend the functionality (and therefore the utility) of the Pablo Instrumentation Environment. This can be done by implementing a coordinated set of functions and data structures to produce special-purpose trace records and event handling. Four trace library extensions which may be of special interest to the user are provided as source code in the `Extensions/Portable` subdirectory of the instrumentation distribution. These extensions are discussed in the following sections and are used by the example presented in appendix A.

4.1 Procedure Tracing

One common application of event tracing is to collect performance data on a procedural basis. That is to say, the instrumented application produces trace data indicating how much execution time each procedure required. This extension of the trace capture library is implemented in the file `ProcTrace.c`.

4.1.1 Procedure Trace Records

One of the first tasks the designer of any extension to the Pablo trace capture library must face is to decide upon the types and content of the special-purpose trace records which are to be generated by the extension. This is primarily an issue of choosing the names and types of the fields that constitute each new kind of trace record, as well as the names and descriptions of those trace records themselves.

The procedure tracing extension accomplishes this by defining a new *family* of events – more precisely, *two* new families, one for procedure entry events and one for procedure exit events. The code which produces the record descriptors for the procedure tracing extension is found in the function `procSDDFdescriptor`, which is invoked twice for each of the two new event families by the function `writeProcRecordDescriptors`. This latter function is the one which will be passed into the trace capture library via the interface routine **setRecordDescriptor** at application run time.

Both the procedure entry and exit full trace records are essentially simple trace records in that they are produced by calls to **traceEvent** only, with no additional application-specific data. These full trace records are supplemented by trace count records, which are produced by calls to **countEvent**. In addition to the fields which should be included in every Pablo Instrumentation Environment trace record (timestamp, event ID, and processor number), both the procedure entry and exit event records generated by this extension include a field called the *procedure index*. This is simply a numerical index into a vector of procedures, designated for tracing in the application. The procedure exit event trace records also include an *inclusive duration* field, indicating the total amount of time elapsed from entry to the indexed procedure until its exit, and an *exclusive duration* field, indicating the amount of time elapsed within the indexed procedure but not within any of its subprocedures that are also being traced by this extension. Recursive procedure calls are naturally supported by nested procedure entry and exit pairs of trace records. The ASCII versions of the full trace record descriptors are shown in figures 17 and 18, the count record descriptors are shown in figures 19 and 20. These appear in the example SDDF trace data file provided in `DataFiles/SDDF/Jacobi-8.BigEndian` (after conversion to ASCII).

```
#97:
// "description" "Procedure Entry Trace Record"
"Procedure Entry Trace" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Procedure" "Procedure Index"
        int     "Procedure Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
};;
```

Figure 17: Procedure Entry Trace Record Descriptor (ASCII)

The `procSDDFdescriptor` function produces these record descriptors by setting up a trace record contents buffer, initializing a pointer to its beginning, and then making a series of calls to the trace library utility functions `sddfWriteInteger` and `sddfWriteString` with the address of this record pointer passed as the first argument, the second being respectively an integer value or the pointer to a string to be written into the record descriptor at that point.[6] An examination of the `procSDDFdescriptor` code and careful reading of the paper *The Pablo Self-Defining Data Format* will make clear the rationale for the sequence of items written into the record descriptors. When all the fields of the record descriptor have been written to the static buffer, the length of the packet is filled in within the buffer and then a `TR_RECORD` type structure (defined in `Trace.h`) is set to point to this buffer. The returned structure is used by `writeProcRecordDescriptors` to pass arguments to the trace capture library internal routine `putBytes` to copy the record descriptor into the trace output buffer itself.

The SDDF definition requires that each trace record have a distinct record name and record tag descriptor value. Although not strictly required, the trace capture library and supplied extensions follow a convention which helps assure that no two trace records have the same tag value. The procedure tracing extension defines two *family tag* values, `FAMILY_PROCENTRY` and `FAMILY_PROCEXIT`, which obey the requirements:

- Use only the bits in `FAMILY_BITMASK`, i.e. none of those in `RECORD_BITMASK`, both of which

[6] The `sddfWriteInteger` and `sddfWriteString` functions are not really trace capture library interface routines *per se*. They are merely simple utility functions in `Internal.c` which copy the integer or string bytes into the character buffer pointed to by the record pointer, then update the record pointer value to point to the next available address. It is the user's responsibility to make certain that the storage addresses are valid.

```
#105:
// "description" "Procedure Exit Trace Record"
"Procedure Exit Trace" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Procedure" "Procedure Index"
        int     "Procedure Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Inclusive Duration" "Inclusive Procedure Duration"
        int     "Inclusive Duration"[];
        // "Inclusive Seconds" "Floating Point Inclusive Duration"
        double  "Inclusive Seconds";
        // "Exclusive Seconds" "Floating Point Exclusive Duration"
        double  "Exclusive Seconds";
        // "Exclusive Duration" "Exclusive Procedure Duration"
        int     "Exclusive Duration"[];
};;
```

Figure 18: Procedure Exit Trace Record Descriptor (ASCII)

```
#98:
// "description" "Procedure Entry Count Record"
"Procedure Entry Count" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Procedure" "Procedure Index"
        int     "Procedure Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Count" "Event Count"
        int     "Event Count";
};;
```

Figure 19: Procedure Entry Count Record Descriptor (ASCII)

```
#106:
// "description" "Procedure Exit Count Record"
"Procedure Exit Count" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Procedure" "Procedure Index"
        int     "Procedure Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Inclusive Duration" "Inclusive Procedure Duration"
        int     "Inclusive Duration"[];
        // "Inclusive Seconds" "Floating Point Inclusive Duration"
        double  "Inclusive Seconds";
        // "Exclusive Seconds" "Floating Point Exclusive Duration"
        double  "Exclusive Seconds";
        // "Exclusive Duration" "Exclusive Procedure Duration"
        int     "Exclusive Duration"[];
        // "Count" "Event Count"
        int     "Event Count";
};;
```

Figure 20: Procedure Exit Count Record Descriptor (ASCII)

are defined in `TraceParam.h`; and

- Do not conflict with any other family tag value in use, either in the `TraceParam.h` file or in any other trace library extension employed by the application.

The trace record tag (both for the record descriptor and corresponding trace data records) is then formed from from the bitwise "or" of the family tag value and the record type (this is the same record type discussed in section 3 in conjunction with figure 7).

The process for generating trace data records themselves is even simpler. The procedure tracing extension code defines structures, each member corresponding in order (and by type) to the fields defined by the proper record descriptor.[7] The event record function `procEventRecord` populates the elements of this structure from data structures specific to procedure tracing and from arguments passed to it (see the section 3 discussion of the **setEventRecordFunction** interface routine for definition of event record functions) and returns the record data pointer.

4.1.2 Procedure Tracing Initialization

The procedure tracing extension to the Pablo trace capture library provides an interface routine, **initProcTrace**, which accepts as arguments the number of procedures being traced by the application and two vectors which contain the event IDs that are to be used for entry to and exit from each procedure, respectively. The two vectors define a matching between procedure entry and exit event IDs, and also define the procedure indices. As a consequence, no event ID may be used more than once. The **initProcTrace** routine uses these parameters to initialize data structures specific to the procedure tracing extension. For each of these event IDs, **initProcTrace** invokes **setEventRecordFunction** so that the trace capture library knows to use `procEventRecord` as that event ID's record function. Also, **setRecordDescriptor** is called so that the trace library will invoke `writeProcRecordDescriptors` at library initialization time.

A second, optional initialization interface function is also provided, **preInitProcTrace**. As its name suggests, it may be invoked before **initProcTrace**. Its use does not replace the required call to **initProcTrace**, but supplements it. The precise reason for the existence of this "pre-initialization" function is esoteric and has to do with how the instrumentation library initializes itself and the required format of the output trace file. The SDDF (self-defining data format) standard requires that all trace data records appearing in a trace file must be preceded by a special record called a *descriptor* and that all these descriptors must be grouped together at the beginning of the file, before the first trace data record. As described in section 3, the trace capture library supplies a set of internal events. Each of these events has its own very special type of record and consequently must have a corresponding record descriptor. Most users will not be interested in these trace library internal events and would find it desirable to have internal event record descriptors appear after all other trace record descriptors. However, the trace capture library must initialize itself when an application first invokes an interface function involving a specific event ID, such as **setEventRecordFunction**. As part of this initialization, the instrumentation library produces the record descriptors for all trace record types declared by the application up to this point (via the **setRecordDescriptor** interface function) followed by the internal event record descriptors. If the application is using more than one trace library extension *and* if the internal event record

[7] The first few elements – packet length, type, and tag descriptor – *must* appear in all data records, by the SDDF standard.

descriptors are to appear in the trace file after all other record descriptors, then each extension must perform some kind of pre-initialization which performs only activities not causing the trace capture library to fully initialize itself, and specifically including the **setRecordDescriptor** invocation for the extension. This is the sole reason for using trace capture library extension pre-initialization functions such as **preInitProcTrace**.

4.2 Message Tracing

Another extension to the Pablo trace capture library produces trace data records for execution of common message-passing interface routines. In implementation, this extension is much simpler than the procedure tracing extension, but demonstrates a different type of interface. The portable portion of this extension is found in the file `MesgTrace.c` with Intel iPSC-specific interface functions in `iPSCtrace.c` and Thinking Machines CMMD-specific interface functions in `CMMDtrace.c`.

4.2.1 Message Passing Trace Records

The message-passing trace extension defines five new families of events: synchronous (blocking) send, synchronous receive, asynchronous (nonblocking) send, asynchronous receive, and miscellaneous global operations. As in the procedure tracing extension, this extension defines functions to produce the record descriptors for each of these families, `writeMesgRecordDescriptors` and `mesgSDDFdescriptor`. The trace records are all full trace records, but in this extension the arguments to the message-passing interface routines (message type and length, also destination node and destination process ID for message send events) are included as separate fields of the trace records. Thus the trace output data provides a nearly complete record of the message-passing activity for the application instrumented with this extension of the trace capture library. The ASCII versions of these trace record descriptors are shown in figures 21, 22, 23, 24, 25, 26, 27, 28, 29, and 30. These also appear in the example SDDF trace data file `Jacobi-8.BigEndian`.

This extension further defines structures for each kind of trace record used, members of which correspond to the fields described by the record descriptors. The ten event record functions:

- `syncSendBeginEventRecord`
- `syncRecvBeginEventRecord`
- `asyncSendBeginEventRecord`
- `asyncRecvBeginEventRecord`
- `mesgOpBeginEventRecord`
- `syncSendEndEventRecord`
- `syncRecvEndEventRecord`
- `asyncSendEndEventRecord`
- `asyncRecvEndEventRecord`
- `mesgOpEndEventRecord`

```
#129:
// "description" "Blocking Message Send Begin"
"Blocking Send Begin" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Type" "Message Type"
        int     "Message Type";
        // "Length" "Message Length"
        int     "Message Length";
        // "DestNode" "Destination Node"
        int     "Destination Node";
        // "PID" "Destination PID"
        int     "Destination PID";
};;
```

Figure 21: Blocking Message Send Begin Record Descriptor (ASCII)

```
#161:
// "description" "Blocking Message Send End"
"Blocking Send End" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Message Seconds" "Floating Point Message Duration"
        double  "Message Seconds";
        // "Duration" "Message Duration"
        int     "Message Duration"[];
};;
```

Figure 22: Blocking Message Send End Record Descriptor (ASCII)

```
#137:
// "description" "Blocking Message Receive Begin"
"Blocking Receive Begin" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Type" "Message Type"
        int     "Message Type";
        // "Length" "Message Length"
        int     "Message Length";
};;
```

Figure 23: Blocking Message Receive Begin Record Descriptor (ASCII)

```
#169:
// "description" "Blocking Message Receive End"
"Blocking Receive End" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Message Seconds" "Floating Point Message Duration"
        double  "Message Seconds";
        // "Duration" "Message Duration"
        int     "Message Duration"[];
        // "Source node" "Sending processor number"
        int     "Sending processor number";
};;
```

Figure 24: Blocking Message Receive End Record Descriptor (ASCII)

```
#145:
// "description" "Nonblocking Message Send Begin"
"Nonblocking Send Begin" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Type" "Message Type"
        int     "Message Type";
        // "Length" "Message Length"
        int     "Message Length";
        // "DestNode" "Destination Node"
        int     "Destination Node";
        // "PID" "Destination PID"
        int     "Destination PID";
};;
```

Figure 25: Nonblocking Message Send Begin Record Descriptor (ASCII)

```
#177:
// "description" "Nonblocking Message Send End"
"Nonblocking Send End" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Message Seconds" "Floating Point Message Duration"
        double  "Message Seconds";
        // "Duration" "Message Duration"
        int     "Message Duration"[];
};;
```

Figure 26: Nonblocking Message Send End Record Descriptor (ASCII)

```
#153:
// "description" "Nonblocking Message Receive Begin"
"Nonblocking Receive Begin" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Type" "Message Type"
        int     "Message Type";
        // "Length" "Message Length"
        int     "Message Length";
};;
```

Figure 27: Nonblocking Message Receive Begin Record Descriptor (ASCII)

```
#185:
// "description" "Nonblocking Message Receive End"
"Nonblocking Receive End" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Message Seconds" "Floating Point Message Duration"
        double  "Message Seconds";
        // "Duration" "Message Duration"
        int     "Message Duration"[];
};;
```

Figure 28: Nonblocking Message Receive End Record Descriptor (ASCII)

```
#193:
// "description" "Miscellaneous Message Operation Begin"
"Message Operation Begin" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Operation" "Operation Type"
        int     "Operation Type";
};;
```

Figure 29: Miscellaneous Message Global Operation Begin Record Descriptor (ASCII)

```
#201:
// "description" "Miscellaneous Message Operation End"
"Message Operation End" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Message Seconds" "Floating Point Message Duration"
        double  "Message Seconds";
        // "Duration" "Message Duration"
        int     "Message Duration"[];
        // "Operation" "Operation Type"
        int     "Operation Type";
};;
```

Figure 30: Miscellaneous Message Global Operation End Record Descriptor (ASCII)

populate these structures and use them for the trace record buffer contents. Unlike the procedure tracing extension, the message tracing extension does not use internal data structures to produce the contents of the trace record fields. Instead, the arguments to the message-passing interface routines are passed to their corresponding event record functions by encapsulating these arguments within a structure and then passing the structure as application-specific data, using the other arguments of the **traceEvent** library interface function.

4.2.2 Message Tracing Initialization

This extension provides an interface function, **initMesgTrace**, to initialize the extension for the instrumented application. Its arguments are:

- An integer vector containing the five event IDs to be used for the begin events: synchronous sends, synchronous receives, asynchronous sends, asynchronous receives, and miscellaneous global operations, in that order.
- An integer vector containing the five event IDs to be used for the end events: synchronous sends, synchronous receives, asynchronous sends, asynchronous receives, and miscellaneous global operations, in that order.

No event ID may be used more than once. For each of these event IDs, **initMesgTrace** invokes **setEventRecordFunction**, passing the appropriate one of the ten above-mentioned event record functions. Also, **setRecordDescriptor** is called so that the trace library will invoke `writeMesgRecordDescriptors` at library initialization time.

This trace library extension, like the procedure tracing extension, also provides an optional "pre-initialization" interface function, **preInitMesgTrace**. The reasons for its existence and its usage are exactly the same as for the procedure tracing extension's pre-initialization function.

4.2.3 Intel iPSC Message Tracing

The file `iPSCtrace.c` contains trace interface functions meant to substitute for the actual blocking and nonblocking message send and receive routines, specifically intended for use with the above message-passing trace extension to the Pablo trace capture library. By invoking **traceCSEND**, **traceCRECV**, **traceISEND** or **traceIRECV** in place of **csend**, **crecv**, **isend** or **irecv**, respectively, and passing the same arguments, the substituted interface function will bundle up the arguments into the required structure, invoke **traceEvent**, and then call the actual message-passing routine, followed by a final **traceEvent** call for the end event. This code assumes exactly one pair of event IDs for each message-passing interface function (e.g. `csendBeginID` contains the blocking send begin event ID). Instead of calling **initMesgTrace**, the application may instead call **iPSCtraceInit** with no arguments. The file `Extensions/SystemDepend/Intel/Include/iPSCtrace.h` may be included in application code to simplify turning message tracing on and off, either by defining or undefining `MESGTRACE`.

4.2.4 Thinking Machines CMMD Message Tracing

The file `CMMDtrace.c` contains trace interface functions meant to substitute for the actual blocking and nonblocking message send and receive routines, specifically intended for use with the

above message-passing trace extension to the Pablo trace capture library. By invoking **traceSENDSYNC**, **traceRECVSYNC**, **traceSENDASYNC** or **traceRECVASYNC** in place of **CMMD_send_block**, **CMMD_receive_block**, **CMMD_send_async** or **CMMD_receive_async**, respectively, and passing the same arguments, the substituted interface function will bundle up the arguments into the required structure, invoke **traceEvent**, and then call the actual message-passing routine, followed by a final **traceEvent** call for the end event. This code assumes exactly one pair of event IDs for each message-passing interface function (e.g. `sendSyncBeginID` contains the blocking send begin event ID). Instead of calling **initMesgTrace**, the application may instead call **CMMDtraceInit** with no arguments. The file `Extensions/SystemDepend/CM5/Include/CMMDtrace.h` may be included in application code to simplify turning message tracing on and off, either by defining or undefining `MESGTRACE`.

4.3 Loop Tracing

Another common application of event tracing is to collect performance data for iteration statements, or loops. That is to say, the instrumented application produces trace data indicating how much execution time each loop required. This extension of the trace capture library is implemented in the file `LoopTrace.c`. This extension is very similar to the procedure tracing extension, with the difference that no bookkeeping is done to relate the execution time of subloops to loops which contain them. Specifically, no distinction is made between inclusive and exclusive durations; only inclusive loop durations are reported.

4.3.1 Loop Trace Records

The loop tracing extension defines two families of events, one for loop entry events and one for loop exit events. The code which produces the record descriptors for the loop tracing extension is found in the function `loopSDDFdescriptor`, which is invoked twice for each of the two event families by the function `writeLoopRecordDescriptors`. This latter function is passed into the trace capture library via the interface routine **setRecordDescriptor** at application run time.

Both the loop entry and exit full trace records are essentially simple trace records in that they are produced by calls to **traceEvent** only, with no additional application-specific data. These full trace records are supplemented by trace count records, which are produced by calls to **countEvent**. In addition to the fields which should be included in every Pablo Instrumentation Environment trace record (timestamp, event ID, and processor number), both the loop entry and exit event records generated by this extension include a field called the *loop index*. This is simply a numerical index into a vector of loops, designated for tracing in the application. The loop exit event trace records also include a *loop duration* field, indicating the total amount of time elapsed from entry to the indexed loop until its exit. The ASCII versions of the full trace record descriptors are shown in figures 31 and 32, the count record descriptors are shown in figures 33 and 34.

4.3.2 Loop Tracing Initialization

The loop tracing extension to the Pablo trace capture library provides an interface routine, **initLoopTrace**, which accepts as arguments the number of loops being traced by the application and two vectors which contain the event IDs that are to be used for entry to and exit from each loop, respectively. The two vectors define a matching between loop entry and exit event IDs, and also

```
#113:
// "description" "Loop Entry Trace Record"
"Loop Entry Trace" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Loop" "Loop Index"
        int     "Loop Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
};;
```

Figure 31: Loop Entry Trace Record Descriptor (ASCII)

define the loop indices. As a consequence, no event ID may be used more than once. The **initLoopTrace** routine uses these parameters to initialize data structures specific to the loop tracing extension. For each of these event IDs, **initLoopTrace** invokes **setEventRecordFunction** so that the trace capture library knows to use `loopEventRecord` as that event ID's record function. Also, **setRecordDescriptor** is called so that the trace library will invoke `writeLoopRecordDescriptors` at library initialization time.

A second, optional initialization interface function is also provided, **preInitLoopTrace**. As its name suggests, it may be invoked before **initLoopTrace**. Its use does not replace the required call to **initLoopTrace**, but supplements it. The reasons for this pre-initialization function and its usage are identical to that described for the procedure tracing extension above.

4.4 Input/Output Tracing

Yet another extension to the Pablo trace capture library produces trace data records for execution of common input/output interface routines. The portable portion of this extension is found in the file `IOtrace.c` with Intel Paragon-specific interface functions in `IntelIOTrace.c` and BSD UNIX-specific interface functions in `UnixIOTrace.c`.

4.4.1 Basic Input/Output Trace Records

The I/O trace extension defines several new families of events, among which are read, write, seek, open, and close. As in the procedure tracing extension, this extension defines functions to produce the record descriptors for each of these families. The trace records are all full trace records, but

```
#121:
// "description" "Loop Exit Trace Record"
"Loop Exit Trace" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Loop" "Loop Index"
        int     "Loop Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Duration" "Loop Duration"
        int     "Loop Duration"[];
        // "Loop Seconds" "Floating Point Loop Duration"
        double  "Loop Seconds";
};;
```

Figure 32: Loop Exit Trace Record Descriptor (ASCII)

```
#114:
// "description" "Loop Entry Count Record"
"Loop Entry Count" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Loop" "Loop Index"
        int     "Loop Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Count" "Event Count"
        int     "Event Count";
};;
```

Figure 33: Loop Entry Count Record Descriptor (ASCII)

```
#122:
// "description" "Loop Exit Count Record"
"Loop Exit Count" {
        // "Time" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Processor number"
        int     "Processor Number";
        // "Loop" "Loop Index"
        int     "Loop Index";
        // "Byte" "Source Byte Offset"
        int     "Source Byte";
        // "Line" "Source Line Number"
        int     "Source Line";
        // "Duration" "Loop Duration"
        int     "Loop Duration"[];
        // "Loop Seconds" "Floating Point Loop Duration"
        double  "Loop Seconds";
        // "Count" "Event Count"
        int     "Event Count";
};;
```

Figure 34: Loop Exit Count Record Descriptor (ASCII)

```
#6001:
// "description" "IO Read"
"Read" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Duration" "Floating Point Duration"
        double  "Duration";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "FP" "File Pointer"
        int     "File Pointer";
        // "Bytecount" "Number Bytes"
        int     "Number Bytes";
        // "Varcount" "Number Variables"
        int     "Number Variables";
        // "Cause" "IO Cause"
        int     "Cause";
};;
```

Figure 35: Input/Output Read Record Descriptor (ASCII)

in this extension the arguments to the I/O interface routines are included as separate fields of the trace records. Thus the trace output data provides a record of the I/O activity for the application instrumented with this extension of the trace capture library.

- The ASCII version of the trace record descriptor for the **read** family of I/O events is shown in figure 35. This record may be generated by substituting invocations in C programs of **read** with calls to **traceREAD**, which accepts the same arguments and returns the same type and value. This wrapper function replaces the original I/O call, invoking it in between generating read begin and end events. A similar wrapper function, **traceFREAD**, is supplied to be used in place of **fread**. Alternatively, an application may invoke the C function

  ```
  traceReadBegin(int filepointer, int numVariables, int cause)
  ```

  before executing the read operation and then

  ```
  traceReadEnd(int nbytes)
  ```

```
#7001:
// "description" "IO Write"
"Write" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Duration" "Floating Point Duration"
        double  "Duration";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "FP" "File Pointer"
        int     "File Pointer";
        // "Bytecount" "Number Bytes"
        int     "Number Bytes";
        // "Varcount" "Number Variables"
        int     "Number Variables";
        // "Cause" "IO Cause"
        int     "Cause";
};;
```

Figure 36: Input/Output Write Record Descriptor (ASCII)

after completing it. The latter two functions have Fortran subroutine counterparts included in the I/O extension to the Pablo trace library, **tracereadbegin** and **tracereadend**. These accept the same arguments as their C language eponyms.

- The ASCII version of the trace record descriptor for the **write** family of I/O events is shown in figure 36. This record may be generated by substituting invocations in C programs of **write** with calls to **traceWRITE**, which accepts the same arguments and returns the same type and value. This wrapper function replaces the original I/O call, invoking it in between generating write begin and end events. A similar wrapper function, **traceFWRITE**, is supplied to be used in place of **fwrite**. Alternatively, an application may invoke the C function

  ```
  traceWriteBegin(int filepointer, int numVariables, int cause)
  ```

  before executing the write operation and then

  ```
  traceWriteEnd(int nbytes)
  ```

```
#5001:
// "description" "IO Seek"
"Seek" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Duration" "Floating Point Duration"
        double  "Duration";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "FP" "File Pointer"
        int     "File Pointer";
        // "Offset" "Offset"
        int     "Offset";
        // "Ptrname" "Ptrname"
        int     "Ptrname";
};;
```

Figure 37: Input/Output Seek Record Descriptor (ASCII)

after completing it. The latter two functions have Fortran subroutine counterparts included in the I/O extension to the Pablo trace library, **tracewritebegin** and **tracewriteend**. These accept the same arguments as their C language eponyms.

- The ASCII version of the trace record descriptor for the **seek** family of I/O events is shown in figure 37. This record may be generated by substituting invocations in C programs of **lseek** with calls to **traceLSEEK**, which accepts the same arguments and returns the same type and value. This wrapper function replaces the original I/O call, invoking it in between generating seek begin and end events. A similar wrapper function, **traceFSEEK**, is supplied to be used in place of **fseek**. In future releases, an application will be able to invoke the C function

  ```
  traceSeekBegin(int fileid, long offset, int whence)
  ```

  before executing the seek operation and then

  ```
  traceSeekEnd()
  ```

  after completing it. The latter two functions will then have Fortran subroutine counterparts included in the I/O extension to the Pablo trace library, **traceseekbegin** and **traceseekend**.

```
#8001:
// "description" "IO Open"
"Open" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Total Time" "IO Time"
        double  "Total IO Time";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "ID" "File ID"
        int     "File Identifier";
        // "Flags" "Flags"
        int     "Flags";
        // "Mode" "Mode"
        int     "Mode";
        // "Name" "File Name"
        char    "File Name"[];
};;
```

Figure 38: Input/Output Open Record Descriptor (ASCII)

- The ASCII version of the trace record descriptor for the **open** family of I/O events is shown in figure 38. This record may be generated by substituting invocations in C programs of **open** with calls to **trace2OPEN** or **trace3OPEN**. The former should be used when only two arguments are passed to **open**, the latter when the optional file creation mode argument is needed. Both functions return the same value and type as **open** when passed the same arguments. This wrapper function replaces the original I/O call, invoking it in between generating open begin and end events. In future releases, an application will be able to invoke the C function **traceOpenBegin** before executing the open operation and then **traceOpenEnd** after completing it. The latter two functions will then have Fortran subroutine counterparts included in the I/O extension to the Pablo trace library, **traceopenbegin** and **traceopenend**.

- The ASCII version of the trace record descriptor for the **fopen** family of I/O events is shown in figure 39. This record may be generated by substituting invocations in C programs of **fopen** with calls to **traceFOPEN**, which accepts the same arguments and returns the same type and value. This wrapper function replaces the original I/O call, invoking it in between generating fopen begin and end events. This interface function and record type will likely disappear in some future release.

```
#3001:
// "description" "IO FOpen"
"FOpen" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Total Time" "IO Time"
        double  "Total IO Time";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "ID" "File ID"
        int     "File Identifier";
        // "Name" "File Name"
        char    "File Name"[];
        // "Mode" "Mode"
        char    "Mode"[];
};;
```

Figure 39: Input/Output Fopen Record Descriptor (ASCII)

```
#1101:
// "description" "IO File Close"
"Close" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "Total Time" "IO Time"
        double  "Total IO Time";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "File ID" "File ID"
        int     "File Identifier";
};;
```

Figure 40: Input/Output Close Record Descriptor (ASCII)

- The ASCII version of the trace record descriptor for the `close` family of I/O events is shown in figure 40. This record may be generated by substituting invocations in C programs of `close` with calls to `traceCLOSE`, which returns the same value and type as `close` when passed the same arguments. This wrapper function replaces the original I/O call, invoking it in between generating close begin and end events. A similar wrapper function, `traceFCLOSE`, is supplied to be used in place of `fclose`. In future releases, an application will be able to invoke the C function `traceCloseBegin` before executing the close operation and then `traceCloseEnd` after completing it. The latter two functions will then have Fortran subroutine counterparts included in the I/O extension to the Pablo trace library, `traceclosebegin` and `tracecloseend`.

4.4.2 Summary Input/Output Trace Records

A detailed trace of every I/O operation with the information described above would obviously allow one to characterize the application I/O. However, the amount of data generated could potentially be extremely intrusive. I/O access patterns can be condensed along dimensions of time and space to reduce the amount of trace data generated. The simplest approach is to reduce trace data generated per unit time by generating records summarizing the I/O activity per file per processor at specified intervals. This will provide enough detail to generate histograms of I/O activity over time. To avoid losing all spatial information, one could output the lowest and highest file byte address that was accessed.

An analogous approach to summarizing the spatial access pattern is to group I/O requests according to the area of a file they are accessing. An even less detailed, although sometimes more

appropriate, method of characterizing the spatial access pattern is to group accesses according to file. In this mode, a record is output for each file close operation. Finally, one may ignore spatial access patterns altogether and consider histograms of statistics such as read and write request sizes, access times, and so on, output over all files in the computation at designated intervals. This provides information in a very condensed form. To this end, the I/O extension to the Pablo instrumentation library supplies interface functions which deactivate generation of the above detailed I/O event records, substituting summary records instead.

- The ASCII version of the trace record descriptor for the `file lifetime summary` family of I/O events is shown in figure 41. This summary mode is activated by invoking **doFileSummaries()**. At present the implementation of this mode is limited to buffered UNIX I/O operations.

- The ASCII version of the trace record descriptor for the `time summary` family of I/O events is shown in figure 42. Time summary statistics may be activated, for a particular window size, by invoking `doTimeSummaryStatistics(double seconds)`. The window size may be changed at any point thereafter with `setTimeWindowSize(double seconds)`.

- The ASCII version of the trace record descriptor for the `block summary` family of I/O events is shown in figure 43. Block summary statistics may be activated, for a particular virtual block size, by invoking `doBlockSummaries(int blockSize)`. The block size may be changed at any point thereafter with `setBlockSummarySize(int blockSize)`.

- The ASCII version of the trace record descriptor for the `histogram` family of I/O events is shown in figure 44. Histogram mode may be activated by invoking:

  ```
  initializeHistograms(lo, hi, numberbins, windowSize, flags)
  int lo, hi, numberbins, flags;
  double windowSize;
  ```

  The recognized `flags` are currently limited to `READ_SIZE_HIST` and `WRITE_SIZE_HIST`. Histograms currently work only with UNIX buffered I/O. To force a dump of histograms corresponding to certain flags, invoke `outputHistograms(int flags)`.

4.4.3 Intel Paragon Input/Output Tracing

A set of routines to generate the basic I/O trace records discussed above for I/O primitives specific to the Intel Paragon is currently under development. All are wrapper functions of the sort mentioned above, i.e. they accept the same arguments and return the same types and values as their namesakes. A list of these interface functions follows:

- `traceCREAD`
- `traceCWRITE`
- `traceIREAD`
- `traceIWRITE`

```
#1201:
// "description" "IO File lifetime summary "
"File Lifetime Summary" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "File ID" "File ID"
        int     "File Identifier";
        // "File Lifetime" "File Lifetime"
        double  "File Lifetime";
        // "Overhead" "Overhead"
        double  "Overhead";
        // "Read Frequency" "Read Frequency"
        int     "Read Frequency";
        // "Read numbytes" "Read numbytes"
        int     "Read Number Bytes";
        // "Read Time" "Read Time"
        double  "Read Time";
        // "Write Frequency" "Write Frequency"
        int     "Write Frequency";
        // "Write Number Bytes" "Write Number Bytes"
        int     "Write Number Bytes";
        // "Write Time" "Write Time"
        double  "Write Time";
        // "Seek Frequency" "Seek Frequency"
        int     "Seek Frequency";
        // "Seek Number Bytes" "Seek Number Bytes"
        int     "Seek Number Bytes";
        // "Seek Time" "Seek Time"
        double  "Seek Time";
        // "Name" "File Name"
        char    "File Name"[];
        // "Mode" "Mode"
        char    "Mode"[];
};;
```

Figure 41: Input/Output File Lifetime Summary Record Descriptor (ASCII)

```
#1301:
// "description" "IO Time Window Summary"
"Time Summary" {
        // "Timestamp" "Timestamp"
        float   "Timestamp";
        // "Duration" "Actual Window Duration"
        float   "Duration";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "File ID" "File ID"
        int     "File Identifier";
        // "Read Frequency" "Read Frequency"
        int     "Read Frequency";
        // "Read numbytes" "Read numbytes"
        int     "Read Number Bytes";
        // "Read Time" "Read Time"
        float   "Read Time";
        // "Write Frequency" "Write Frequency"
        int     "Write Frequency";
        // "Write Number Bytes" "Write Number Bytes"
        int     "Write Number Bytes";
        // "Write Time" "Write Time"
        float   "Write Time";
        // "Seek Frequency" "Seek Frequency"
        int     "Seek Frequency";
        // "Seek Number Bytes" "Seek Number Bytes"
        int     "Seek Number Bytes";
        // "Seek Time" "Seek Time"
        float   "Seek Time";
        // "First Byte" "First Byte Accessed"
        int     "First Byte";
        // "Last Byte" "Last Byte Accessed"
        int     "Last Byte";
};;
```

Figure 42: Input/Output Time Summary Record Descriptor (ASCII)

```
#4001:
// "description" "IO Block Summary"
"Block Summary" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "Count" "Event Count"
        int     "Event Count";
        // "Bytes" "Number Bytes"
        int     "Number Bytes";
        // "File" "File ID"
        int     "File ID";
        // "Total Time" "IO Time"
        double  "Total IO Time";
        // "Offset" "Offset"
        int     "Byte Offset";
};;
```

Figure 43: Input/Output Block Summary Record Descriptor (ASCII)

```
#1501:
// "description" "IO Histogram Data"
"Histogram Data" {
        // "Timestamp" "Timestamp"
        int     "Timestamp"[];
        // "Seconds" "Floating Point Timestamp"
        double  "Seconds";
        // "ID" "Event ID"
        int     "Event Identifier";
        // "Node" "Node ID"
        int     "Node Identifier";
        // "Dummy" "Dummy"
        int     "Dummy Int";
        // "Histogram" "Histogram"
        int     "Histogram Data"[];
};;
```

Figure 44: Input/Output Histogram Summary Record Descriptor (ASCII)

- `traceSETIOMODE`
- `traceIOWAIT`

Each of these has a Fortran subroutine or function equivalent by the same name (ignoring upper/lower case).

4.4.4 Input/Output Tracing Initialization

This extension provides an interface function, `IOTraceInit`, to initialize the extension for the instrumented application. As with other extensions discussed above, the initialization function must be called by an application before attempting to use any of the other interface functions provided by the extension. From Fortran the subroutine `iotraceinit` should be called.

This trace library extension, like the procedure tracing extension, also provides an optional "pre-initialization" interface function, `preInitIOTrace`. The reasons for its existence and its usage are exactly the same as for the procedure tracing extension's pre-initialization function.

5 Preparing Trace Files for Visualization

Once an instrumented application has been executed using the Pablo trace capture library, the user will have one or more output trace data files conforming to the binary SDDF specification. There is a small amount of post-processing that must be done before this data may be analyzed using the Pablo Visualization Environment. This is the topic of the sections which follow.

5.1 Merging Trace Files

Particularly when a parallel application has been instrumented, each processor (node) will have generated its own output trace data file (or files). Software is provided to merge any set of binary SDDF trace files into a single file provided the following conditions are met:

- The merging programs mentioned below must be compiled and executable on the machine which is to perform the trace file merging.
- Every trace data record in every one of the files must contain a scalar-valued field with the same name, which will be used as the key field for performing the merge. The data records in each file must appear in nondecreasing order, when the ordering is on the value of this key field.

The `SDDFmerge` program must be compiled and installed on the machine on which the merge is to be done. Source code to this program is part of the Pablo software distribution, found in the file `SDDFlibrary/SamplePrograms/SDDFmerge.c`. The shell script `makeProgram` in that directory may be invoked to compile the program.

Binary SDDF trace data files may be merged by executing the following command:

`SDDFmerge` *list of trace data files*

The merged output is sent to the standard output stream and should be redirected to a file. Alternatively, merged trace data may be sent to a file by specifying the option and argument `-o` *output file name* to the `SDDFmerge` program before the list of files to be merged.

By default `SDDFmerge` uses the field name `Seconds` as the merge key field name. This may be overridden by specifying the option and argument `-k` *key field name* to the `SDDFmerge` program before the list of files to be merged.

Also by default `SDDFmerge` uses `MergeTwoFiles` as the name of the program to execute to merge each pair of binary SDDF files. This may be overridden by specifying the option and argument `-m` *merge program name* to the `SDDFmerge` program before the list of files to be merged. The pairwise merge program must appear in the invoking user's execute search path and be executable by the user. Source code to the `MergeTwoFiles` program may be found in the Pablo distribution in the file `SDDFlibrary/SamplePrograms/MergeTwoFiles.C`.

5.2 Conversion of Trace Files

Once a single trace data output file has been obtained, it may be converted to ASCII by executing the `SDDFconverter` program. The source code for this program is found in the same directory as the `SDDFmerge.c` and `MergeTwoFiles.C` source files. The `SDDFconverter` program is interactive, prompting the user for input and output file names. The advantage of having an ASCII format file

is that they are more easily inspected by human eyes. The Pablo Visualization Environment may directly use the binary SDDF file.

A A Representative Example

The Pablo Instrumentation Environment includes source code for an extended example demonstrating application of most of the concepts introduced in this guide, including effective use of two of the extensions to the trace capture library presented in section 4. Code for this example is found in the directory `Tracelib/Examples/Intel/PDE`. Although the application is specific to the Intel iPSC systems, the lessons to be learned in examining how it exploits the features of the trace library are not system-specific.

A.1 Application Makefile

The `Makefile` in the `PDE` directory is an excellent example of how such a file may be constructed for other applications. It shows conditional compilation of the application with neither, one, or both of the procedure and message tracing extensions through the use of the compiler flags `PROCTRACE` and `MESGTRACE`. It also shows inclusion of both portable and system-dependent trace library header files within application code, and linking of the application with the trace library itself. In preparing a new application for instrumentation with the Pablo Instrumentation Environment, the user would be well advised to begin with a `Makefile` such as this and modify it as needed.

A.2 Host Application Code

The file `pdeHost.c` does not itself get compiled with any part of the trace library, but it does perform a few functions vital to the iPSC system-specific implementation of the trace library. First, it invokes **setpid** to set the process ID of the host process, absolutely essential before using any message-passing routines. (This assumes the user has done a **getcube** before starting the example application.) Second, it **load**s the same node code onto all nodes in the current partition, assigning them all the same process ID. Finally, it performs a **waitall** for all nodes in the partition running the node process code.

Each system-specific implementation of the trace capture library will likely have requirements similar to this. These are all requirements placed on Intel iPSC applications instrumented with the Pablo trace library.

A.3 Node Application Code

A.3.1 Instrumentation Initialization

If the `PROCTRACE` compiler flag is turned on, the node code defines two vectors holding the procedure entry and exit event IDs. These are passed to **initProcTrace** shortly after the application begins executing.

If the `MESGTRACE` compiler flag is turned on, the node code defines the event IDs used for message-passing interface routines, matching the declarations in the `iPSCtrace.c` extension file.

If either of the tracing compiler flags is turned on, the application invokes **setTraceFileName**. Note the inclusion of the processor number within the trace output file name.

A.3.2 Run-Time Instrumentation

If the `PROCTRACE` compiler flag is turned on, then immediately before the invocation of each procedure to be traced, the node code calls **traceEvent** to produce a procedure entry simple trace record. Likewise, **traceEvent** is called immediately after the procedure returns, generating a procedure exit record.

If the `MESGTRACE` compiler flag is turned on, then every invocation of a message-passing interface function in the node code is replaced with a call to the substitute interface function provided in `iPSCtrace.c`.

A.3.3 Trace Data Collection

Before terminating, if either of the tracing compiler flags is turned on, the node code invokes **endTracing**. When the entire application has completed, the resulting trace data files, one for each processor used, are merged using the techniques found in section 5. The merged trace data file thus obtained, when run through the `SDDFconverter` program, produces an ASCII SDDF file.