OSF–RI MK7
The Event Trace Analysis Package: ETAP Users Guide

By
Joseph CaraDonna
Revision 0.7 - February 17, 1995

DRAFT

RESEARCH INSTITUTE
11 Cambridge Center
Cambridge, MA 02142

TABLE OF CONTENTS

TABLE OF CONTENTS

Introduction

The Event Trace Analysis Package (ETAP) is an OSF Mach tracing facility, designed to provide users with a robust set of tools to study system behavior. ETAP supports both kernel internal and application level event tracing. Kernel event tracing provides system programmers with a powerful tool for monitoring kernel internal activity. Application level trace support provides application programmers with a consistent, native to Mach, model for application specific event tracing. Together, the two levels of tracing enable users to better understand the interaction between their applications and the Mach operating system.

Event Types and Subsystems

ETAP categorizes events by type and subsystem. Event types are used to identify specific system events (i.e., specific kernel lock operations, thread context switches, interrupts). Each event type is represented by a text name and a numerical value. Users may toggle modes of tracing on specific events, using the more intuitive text names as handles. The kernel, however, refers to the event types by their numerical values. Information for each event type can be obtained using the etap_status utility [see etap_status).

ETAP subsystems are used to associate event types of a common subsystem. For example, all lock event types defined in the virtual memory subsystem (such as: vm_map, vm_object and vm_bucket) would belong to the lock_vm subsystem. Subsystems are useful for identifying related events and atomically enabling/disabling the tracing of associated events. A list of configured subsystems can be obtained using the etap_status utility (see etap_status).

Note: The event types and subsystems available depend on the kernel's ETAP configuration.

ETAP Architecture

In order to analyze events in the micro-kernel, it is necessary for the kernel to collect data via a scheme that does not affect the results being tabulated or add significant overhead to the system. Hence, the micro-kernel is instrumented to collect event data and all statistical analysis is performed outside the kernel.

The kernel's instrumentation consists mainly of event probes. Event probes detect ETAP event occurrences and record information, pertaining to the event, in the kernel's ETAP buffers. A set of ETAP utilities is used to fetch the recorded data from the buffers, writing it to disk. Once the traced data is written to disk, ETAP analysis utilities may be used to access the files and perform event analysis (see Figure 1).

ETAP implements two types of data storage buffers: cumulative and monitored. The cumulative buffer is used to store event data which is tallied over time. Cumulative data collection is only used for tracing lock events (discussed later).

A monitored buffer is allocated for each processor on the host machine, and is used to store individual event occurrences. Monitored buffers are circular, where once the buffer has reached maximum capacity it wraps around to the top again. Monitored buffers are used by lock event and thread event tracing.

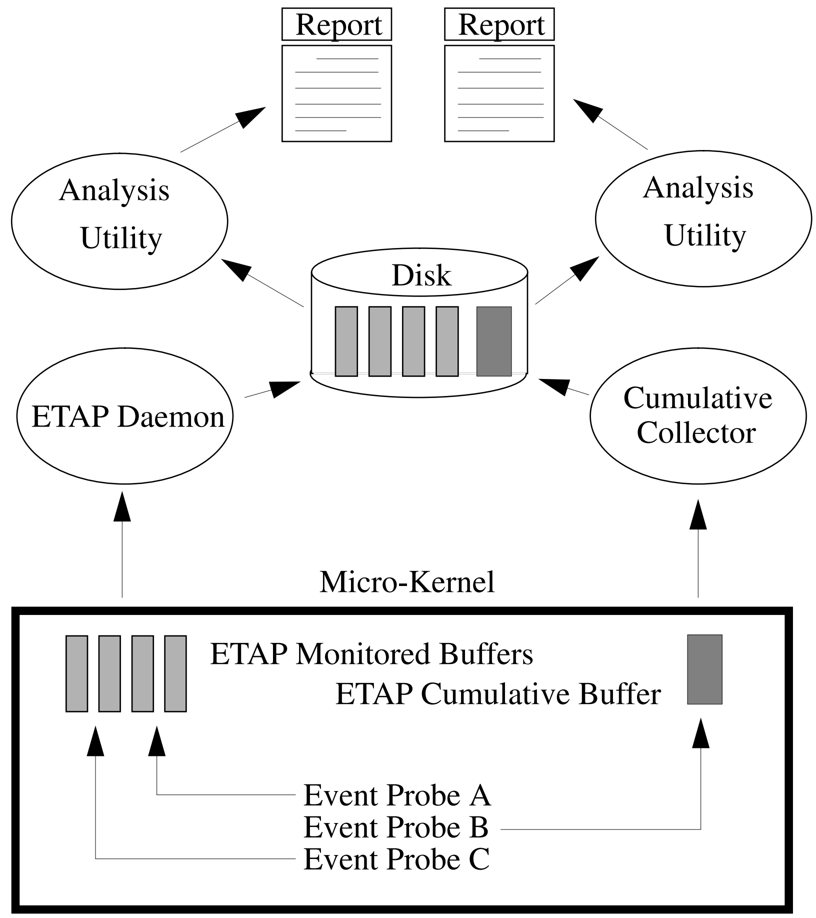


FIGURE 1 **This picture shows the flow of event data from kernel event probes to kernel ETAP buffers to disk. Once the data is written to disk, analysis utilities may generate event reports. Note: four processors are assumed here.**

Obtaining Buffered ETAP Data

Obtaining the buffered ETAP data from the kernel requires two user-level utilities: the ETAP Daemon and the Cumulative Collector. The function of the cumulative collector is straightforward, providing the user with a snapshot of the kernel's ETAP cumulative buffer. It simply maps the cumulative buffer into its address space and writes the contents of the cumulative buffer to a file on disk. A statistical report may be generated from this file using a separate analysis utility.

The ETAP daemon, however, is more involved, providing user access to the kernel's ETAP monitored buffer(s). The ETAP Daemon is a multi-threaded daemon which monitors the activity in the kernel's ETAP monitored buffer(s). Since the kernel may have multiple monitored buffers, the daemon spawns a thread to monitor an assigned buffer. The daemon's threads write event data to disk as new records are stored in the monitored buffer(s). Hence, the daemon's data collection occurs in real-time. If a buffer is not changing, its assigned thread sleeps and periodically wakes up to check the buffer. If the buffers contents have changed, the thread continues its process of writing the data to disk until the buffer stops changing.

As mentioned previously, the ETAP monitored buffers are circular and wrap whenever maximum capacity is reached. Because of this property, new event entries eventually overwrite old ones. Event data is lost when the kernel logs new event occurrences to a buffer faster than an ETAP Daemon thread can process them. Thus, it is important for each thread to record the status of its data collection. ETAP Daemon threads detect overload and record the number of lost records lost so sampling validity can be determined.

Once tracing is complete, the data file(s) produced by the ETAP Daemon, may be analyzed by a separate utility for report generation.

ETAP Instrumentation

The micro-kernel instrumentation is the core of the event trace analysis package. This section focuses primarily on the ETAP event data collection instrumentation.

The ETAP kernel instrumentation supports two modes of tracing:

1. Thread event tracing

2. Lock event tracing

Thread event tracing collects thread execution data supplied by either the kernel internal or user-level event probes. Lock event tracing captures kernel internal lock activity for behavioral study. The following subsections cover each ETAP trace mode in greater detail

Thread Event Tracing

ETAP thread event tracing provides users with thread execution information for both user-level and kernel threads. The kernel has been instrumented with event probes to detect and record ETAP event occurrences.

Event probe data is stored in the kernel's ETAP monitored buffer(s) and consists of the following information:

- event type
- event flags
- timestamp
- program counter
- ID of invoking thread
- event specific data

Event types are unique values which are used to identify specific trace events. Event flags are a set of bit fields which provide event flavor information, such as, whether the event was posted by the kernel or an application (see Appendix A for a complete listing). The timestamp marks where in time the event occurred. The program counter is recorded to identify the instance of the event. Lastly, event specific information is often recorded to better understand the nature of the event (e.g., thread priority).

Application programmers may instrument their programs with "user-level probes" to trace user-level thread activity. A user-level probe is simply a Mach system call, etap_probe(), which acts as an interface to the kernel's ETAP monitored buffer(s). The etap_probe() system call records the same type of information as the kernel event probes (with the exception of the program counter). Event specific data may be passed as a parameter to the system call. See the ETAP System Calls section for more information on etap_probe().

All kernel ETAP probes are defined in greater detail in Appendix A. An example C language program, demonstrating user-level probes, is located in Appendix B.

Thread Trace Control

ETAP thread event tracing provides users with thread execution information for both user-level and kernel threads. It is this information which allows users to see what really happened while the thread executed. Events such as: thread creation, context switches, system calls and thread termination can all be recorded as the thread executes (see Appendix A for a listing of all kernel event probes).

By default, threads are initialized with ETAP event tracing deactivated. Thread trace status may be controlled via the etap_trace_thread() system call. When a threads trace status is active, the selected thread events will be recorded for that thread.

Lock Event Tracing

Lock event tracing is designed to capture kernel internal lock activity for behavioral study. This mode of tracing is primarily used by operating systems developers to analyze kernel lock performance.

The ETAP instrumentation supports two levels of lock event tracing: cumulative and monitored. Cumulative lock event tracing accumulates data for each lock in the kernel and stores the information in the ETAP cumulative buffer. Monitored lock event tracing collects data per lock operation, where each instance of lock duration or contention is individually recorded in the ETAP monitored buffer(s).

The two levels are designed to complement each other, conserving both memory and analysis time. Cumulative event collection can detect locks with poor performance at a low memory cost, but it cannot pinpoint the exact cause of long durations or contention times. Monitored event collection can isolate specific lock operations, but its memory requirements grow significantly as the number of monitored locks increases. The idea is to first perform cumulative lock tracing en masse or on a subsystem, deriving a list of locks suspected of poor performance. Monitored lock tracing can then commence, a few suspect locks at a time.

Cumulative Lock Event Tracing

Cumulative lock event tracing requires minimal support with low overhead. In general, this instrumentation highlights lock behavior patterns and detects locks that require further monitoring.

Lock Duration Instrumentation

The following per lock data aid in determining lock acquisition patterns and the length of time locks are being held:

- The number of times a lock was acquired
- The cumulative time the lock was held
- The cumulative sum of the hold times squared
- The minimum time the lock was held
- The maximum time the lock was held
- The start time for the last lock acquisition, used to determine the length of time the lock was held

From the above data, the following statistical information is derived for each lock:

- The mean time the lock was held

- The standard deviation from the mean

The statistical data pertinent for all types locks, but more relevant for spin locks, since mutex and complex locks may cause threads to block resulting in high hold times. Cumulative tracing should first be enabled and analyzed for spin locks. In particular, if a spin lock has a large mean time, indicating it is usually held for relatively long periods of time, it will be immediately placed on the suspect list of locks to monitor. Additional locks to analyze can be determined by comparing the minimum and maximum lengths of time a lock is held to the mean and standard deviation. If the mean time and standard deviation for a lock are small and the minimum and maximum times are close to the mean under several types of loads, the lock usage patterns for that lock probably do not require further study.

Lock Contention Instrumentation

Lock contention is another area that can easily be analyzed by collecting simple lock event data. The following cumulative data are maintained for each lock event:

- The total number of acquisitions, including attempts at conditional locking
- The number of failed attempts to acquire the lock
- The cumulative time threads waited for acquisition
- The cumulative squares of wait times
- The minimum wait time
- The maximum wait time
- The time the last failed lock attempt began, used to determine the time spent waiting

The number of failed lock attempts is relevant for complex locks as well as for simple locks, since a large number of misses is indicative of a highly contended lock. In addition, simple locks with large values for any of the following require further monitoring: mean wait time, standard deviation from mean wait time, maximum wait time. Locks requiring in-depth study are determined by analyzing the cumulative data for hold and wait times. The selected locks are then monitored more closely to determine the code paths that require restructuring.

Cumulative Interval Data Collection

The cumulative lock tracing instrumentation also collects data on a time interval basis. This feature provides more detail about the actual distribution of lock times.

The micro-kernel instrumentation increments a counter for the appropriate time interval when measuring a lock operation. The number of time intervals is a compile-time configurable option, while the actual width of the intervals may be changed at run-time. The interval instrumentation supplies data points that may be used to determine the shape of the distribution curves for lock usage patterns. In particular, it provides a good approximation of the percentage of lengthy lock operations. Thus, a lock with large values for the higher interval counters may be a significant performance problem and must be monitored. On the other hand, a lock with only a few lengthy utilization times is likely to cause intermittent performance problems. This interval method is applicable to both lock duration and contention times.

Monitored Lock Event Tracing

After creating a list of candidates for further study, selected locks may be monitored more closely. Lock events may be monitored individually, where one or both of the following may be recorded: lock duration or lock contention. This level of lock event tracing records individual lock operations in the kernel's ETAP monitored buffer(s).

Each recorded monitored lock event is fairly small, containing the following information:

- lock event
- event flags
- address of the lock
- time measured
- program counter where the operation began
- program counter where the operation completed

The lock event identifies the lock which triggered the event. The event flags indicate: the type of trace (duration/contention), the type of lock (spin/complex/mutex), and the type of operation (read/write). The lock address is supplied to allow for specific occurrences of a lock to be traced. The time measured is the time it took to execute the operation. The program counter information enables execution code path to be isolated.

Event Trace Control

Event trace control enables users to select events of interest, as well as, configure the level of tracing to be performed. Event selection is useful for filtering-out specific events, keeping data collection and trace skew to a minimum. Each ETAP event can be individually enabled or disabled by users anytime throughout the system's life-span.

ETAP supports a variety of trace configurations, many of which pertain to lock event tracing only. When tracing lock events, the trace configuration may be any of the following:

- cumulative hold, measures cumulative lock duration
- cumulative wait, measures cumulative lock contention
- monitored hold, measures monitored lock duration
- monitored wait, measures monitored lock contention

Note: The supported trace configuration combinations depend on the kernel's ETAP configuration.

Thread events share the same trace configurations as lock events, but the meanings are different. Each trace configuration, when applied to thread events, perform the same function: event trace enable/disable toggling. If any of the trace configurations are enabled for a thread event, then the thread event is enabled. All configurations must be disabled, however, for the thread event to be disabled. By default, the ETAP utilities use monitored hold to toggle thread event trace status.

Event trace control is maintained via two ETAP utilities: etap_trace and etap_status. It is the etap_trace utility which allows users to enable and disable the tracing of ETAP events individually. The etap_status utility displays the current trace status of each ETAP event. The ETAP event trace control utilities are discussed in detail in the ETAP Utilities section.

Note: Event trace control is used in conjunction with thread trace control. Together, they create a two dimensional filter:

Thread events are only recorded if the thread's ETAP trace status is active and the kernel's thread events are enabled.

For example, Figure 2 shows the execution of a thread as it passes kernel ETAP probes. Assume probes P1 and P3 were enabled, while probe P2 is disabled. Once the thread intersects Probe P1, the probe checks to see if the thread's ETAP trace status is enabled. Since the thread's ETAP status is enabled, the probe data is recorded. The thread then intersects Probe P2, but no data is recorded at this point since Probe P2 is disabled (regardless of the thread's trace state). Now assume the thread's ETAP trace status was deactivated (via the etap_trace_thread system call) after P2 but prior to intersection with Probe P3. Once the thread intersects Probe P3, data is not recorded, since the thread's ETAP trace status is inactive (regardless of the probes state). See Appendix B for an example C language program, demonstrating thread trace control usage.

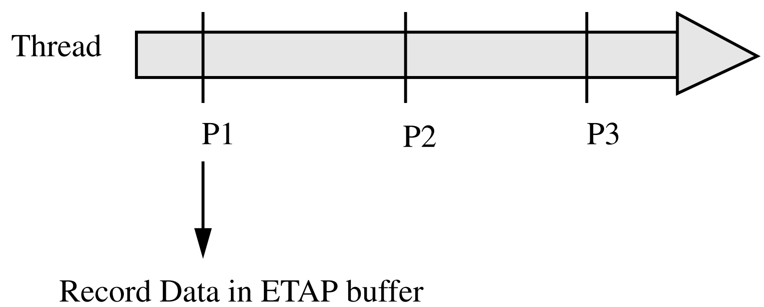
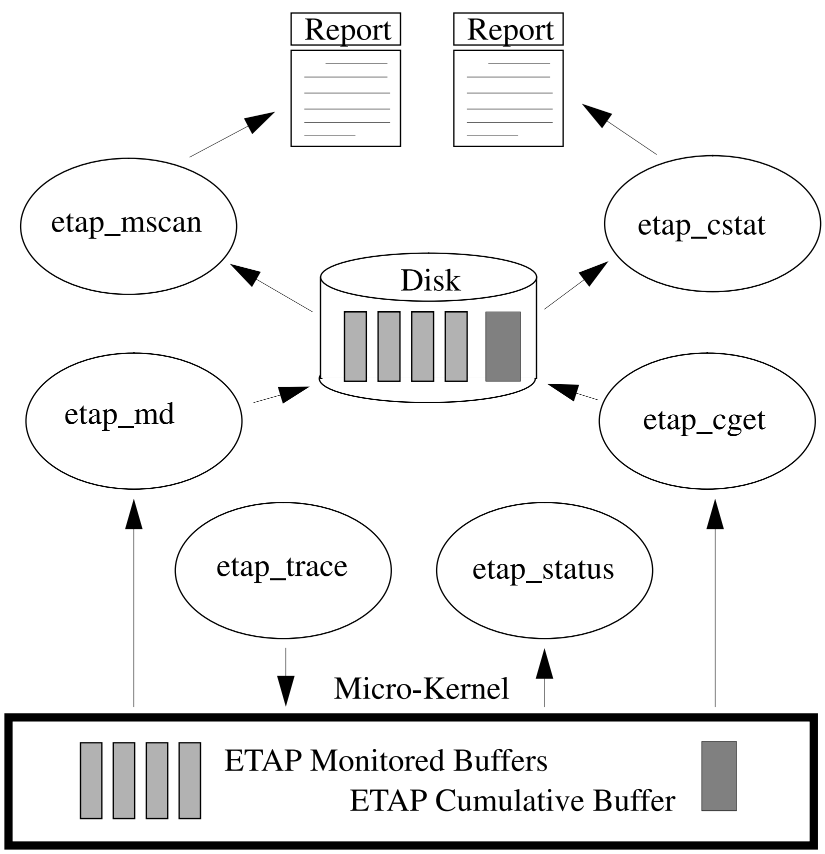


FIGURE 2 **The execution of a thread as it passes kernel ETAP probes.**

ETAP Utilities



The micro-kernel instrumentation described above traces event activity and logs the data in the ETAP kernel buffers. A set of ETAP utilities provides users with trace control, the ability to obtain the buffered data from the kernel, and performance analysis tools.

etap_status

The etap_status utility displays the contents of the kernel's ETAP event and subsystem tables.

The ETAP event table contains a list of configured kernel events (both text and numerical representations) along with the current trace status for each. The trace status of an event may be either on or off in any of the following trace modes: cumulative hold, cumulative wait, monitored hold, monitored wait.

The subsystem table contains a list of configured ETAP subsystems, in both text and numerical representations.

Indirectly, the etap_status utility serves as an on-line directory of all the events and subsystems configured in the micro-kernel. A synopsis of its usage follows:

etap_status [table]

The table switch may be one or both of the following:

-e event table (default)

-s subsystem table

etap_trace

The etap_trace utility is used to enable and disable various modes of event tracing. A synopsis of its usage follows:

etap_trace [mode] [type] [-d] objects

The mode parameter selects the trace operation and may be one of the following:

-m monitored trace (default)

-c cumulative trace

-R ETAP reset

If a mode is not selected at the command line, etap_trace selects the monitored trace mode. Note: the trace modes available depend on the configuration of the kernel. If the reset mode is selected, all event tracing is disabled and the cumulative buffer is cleared. If the cumulative buffer is not configured in the kernel, then only event tracing effected.

The type parameter can be used to either select a trace operation or redefine the cumulative buffer interval width:

-h trace hold time - duration (default)

-w trace wait time - contention

-i# redefine the cumulative buffer interval width

Contention and duration operations are typically associated with the timing of lock events. When tracing general events, such as p_thread_life, either trace operation (hold or wait) may be used. If a type is not selected at the command line, etap_trace defaults to trace hold time.

When using the type parameter to redefine the cumulative buffer interval width, the -i switch is followed by a number specifying the width in microseconds. The cumulative buffer interval width can only be changed when ETAP is being reset (via the -R switch). This restriction prevents the collected distribution interval data from becoming distorted. The -d switch is used to disable tracing on the specified objects. If -d is not selected at the command line, etap_trace enables tracing on the specified objects. The objects parameter is used to specify either a list of event types or subsystems. The options are as follows:

-e event type (lock or kernel probe)

-s subsystem

-a all events

If either event type or subsystem is selected, a text name of the object is expected to follow. Etap_trace accepts multiple instances of -e and -s on the command line.

Usage

To demonstrate the flexibility of the etap_trace utility, several examples of its usage are supplied below. Note: Defaults are used where applicable.

Enable cumulative hold tracing for the vm_map and ipc_object lock events:

```
etap_trace -c -e vm_map -e ipc_object
```

Disable all cumulative lock tracing without clearing the cumulative buffer:

```
etap_trace -c -d -h -w -a
```

Enable monitored tracing of the p_thread_life probe:

```
etap_trace -e p_thread_life
```

Enable monitored wait tracing for all kernel events:

```
etap_trace -w -a
```

Disable all monitored lock event tracing for the virtual memory subsystem:

```
etap_trace -d -s vm
```

Disable all tracing, clear the cumulative buffer and set the cumulative buffer interval width to 120 microseconds:

```
etap_trace -R -i 120
```

etap_config

The etap_config utility displays the current systems ETAP configuration. Upon execution, the following information is shown:

Table Info

- Number of entries in the event table
- Number of entries in the subsystem table

Cumulative Info

- Size of the kernel's cumulative buffer (bytes)
- Size of a cumulative buffer entry (bytes)
- Maximum number of entries the buffer can contain
- Current interval width (microseconds)

Monitored Info

- Size of the kernel's monitored buffer(s) (bytes)
- Size of a monitored buffer entry (bytes)
- Maximum number of entries the buffer can contain
- Number of monitored buffers

If either mode of tracing (cumulative or monitored) is not configured, the etap_config utility will display "NOT CONFIGURED" under the mode info heading.

etap_md - The ETAP Daemon

When ETAP thread event or lock event monitoring is configured in the system, the kernel records traced data in circular per-processor monitored buffers. Etap_md is a multi-threaded daemon that monitors the kernel's monitored buffer(s) and writes the recorded data to disk as event activity persists.

A synopsis is as follows:

etap_md [-o file]

The user may specify a base filename via the -o option; otherwise the base /tmp/etap_mdata.log. As mentioned previously, each processor has its own monitored buffer. Likewise, the etap_md daemon creates a separate file for each buffer. The file name is derived from the base name, where the processor number is appended to the end of the file name prior to the extension. For instance, if the base file name is etap_mdata.log and there are four active processors, the output files would be: etap_mdata0.-log, etap_mdata1.log, etap_mdata2.log and etap_mdata3.log. All monitored buffer files are written in the ETAP Mdata file format (see ETAP Mdata for details).

The etap_md daemon spawns one thread per monitored buffer. Each thread writes the contents of its monitored buffer to disk as events occur, collecting data in real-time. If a buffer is not changing, its associated etap_md thread sleeps and periodically wakes-up to check the buffer. If the contents of the buffer have changed, the thread continues its process of writing the data to disk until buffer activity ceases.

Because the monitored buffers are circular, new event entries eventually overwrite old ones. Data is lost when kernel event trace activity occurs faster then the etap_md daemon can write them out to disk. For this reason, the etap_md daemon tracks data loss and periodically writes the current status out to disk. The default file name of the lost data log is /tmp/etap_mdata.lost. However, if the -o option is selected, the name of the lost data file is derived by appending a .lost extension to the specified base file name. Lost data files are written in the ETAP Mlost file format (see ETAP Mlost).

Once monitored event data is stored on disk, user-level utilities (i.e., etap_mscan and etap_mstat) may view and analyze the data to produce reports.

etap_mscan - Analysis Utility

The etap_mscan is a sample analysis program which scans and displays the contents of Mdata and Mlost files created by the ETAP daemon (etap_md). This program simply opens the files, reads the headers and prints what it sees (somewhat formatted) to standard out. From this simple program, more complex tools can be derived to manage and give meaning to the traced data.

ETAP System Calls

The OSF Mach ETAP subsystem is directly accessible through a small collection of system calls. Each system call is described in detail in the following subsections.

etap_probe

Application programmers may instrument their applications with user-level probes, using the etap_probe() Mach system call. All event data passed to the kernel via etap_probe is recorded in the kernel's ETAP monitored buffer(s).

Synopsis

kern_return_t **etap_probe**

(unsigned short	*event_id,*
unsigned int	*data_size,*
etap_data_t	**data*);

Parameters

event_id

A user defined value used to identify the event.

data_size

The size (in bytes) of the data being passed. Note: The data size is limited to ETAP_DATA_SIZE, which is defined in etap.h.

data

A pointer to the event data being passed.

Library

libmach

Includes

mach/etap.h

etap_trace_thread

The etap_trace_thread() system call is used to toggle the ETAP trace status of a thread.

Synopsis

```
kern_return_t etap_trace_thread
        (mach_port_t        target_thread,
         boolean_t          active);
```

Parameters

target_thread

The port of the thread who's ETAP trace status will be toggled.

active

The boolean value (either TRUE or FALSE) stating whether the thread's ETAP trace status will be activated or not. Passing TRUE will enable the thread's trace status and FALSE will deactivate it.

Library

libmach

Includes

mach/etap.h

How-To

This section should be used as a quick reference guide to get things going. It addresses typical “how to” questions tying things together.

Trace Kernel Events

All kernel event tracing is disabled by default. Kernel events may either be enabled individually or by subsystem.

Steps:

1. Execute the ETAP Daemon in the background (see etap_md)
2. Find events you want to trace (etap_status)
3. Enable tracing for the events of interest (etap_trace)
4. Start application
5. Activate ETAP trace status for threads of interest (etap_trace_thread)
6. Disable tracing for the events of interest (etap_trace)
7. Analyze collected event data (see etap_mscan)

Note: *It is often useful to terminate the ETAP Daemon after each trace session (trace sessions consist of steps 3 through 6). Terminating the ETAP daemon causes the event trace data files to be properly flushed to disk and closed. If the daemon is not terminated between trace sessions the data files it* *created will continue to grow, containing data from multiple trace sessions.*

Trace Application Events

The Event Trace Analysis Package is designed to support the tracing of application specific events. Applications are instrumented using the etap_probe system call (see ETAP System Calls).

All logged events are stored in the kernel's ETAP monitored buffer(s) where the ETAP Daemon retrieves and writes the data to disk.

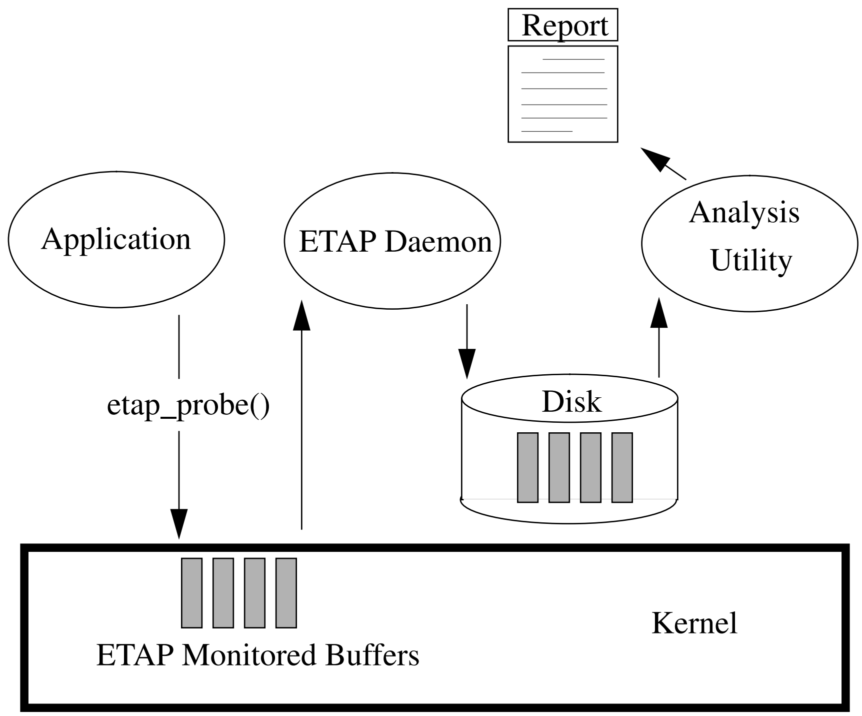


FIGURE 3 **This picture shows the flow of application specific event data from the application to report form.**

Steps:

1. instrument application (see ETAP System Calls)
2. execute the ETAP Daemon in the background (see etap_md)
3. enable the p_user_events thread event probe (see etap_trace)
4. execute application
5. terminate the ETAP Daemon (optional)
6. disable p_user_events (see etap_trace)
7. analyze collected event data (see etap_mscan)

ETAP Configuration

The event trace analysis package requires both the kernel and the server to be configured with ETAP options enabled.

Kernel Configuration

The kernel's ETAP configuration options are:

1. Thread event tracing: ETAP_EVENT_MONITOR

Thread event tracing enables the kernel internal, as well as, user-level thread events to be traced. At the kernel level, information such as thread creation, termination and context switches can be recorded in the monitored buffer. At the user-level, application specific events may be recorded in the kernel's monitored buffer(s).

2. Monitored lock event tracing: ETAP_LOCK_MONITOR

Monitored lock event tracing collects lock event data for each lock operation. When enabled, each instance of lock acquisition or contention can be individually recorded in the kernel's monitored buffer.

3. Cumulative lock event tracing: ETAP_LOCK_ACCUMULATE

Cumulative lock event tracing accumulates data for each lock in the kernel. When enabled, lock operation data may be accumulated in a dedicated per-lock cumulative buffer entry.

Each of these trace options are mutually exclusive. To enable the trace package, the ETAP switch along with ONE ETAP trace option must be selected. All ETAP kernel options are located in the kernel's config.debug file.

The selected ETAP option determines the level of kernel instrumentation built into the kernel. Option 1 enables event probes throughout the system, while options 2 and 3 add instrumentation to the kernel lock mechanisms.

ETAP (and all its options) is mutually exclusive with the MACH_LDEBUG option. It is assumed that general lock debugging is completed before gathering event trace statistics.

ETAP options are normally only enabled for event profiling and performance studies. Event tracing should not be enabled for release configurations, as the code size and performance impact of these options are significant.

Server Configuration

Even though the server itself is not instrumented, ETAP must be enabled in the server configuration for the ETAP package to function as a whole. The server is responsible for performing the mapping of kernel ETAP buffers and tables to the ETAP user-level utilities upon request.

Server configuration is straight forward, simply enable the ETAP option in the server's config.common file. Upon server boot-up, the ETAP configuration will be displayed on the console. If the ETAP package was configured properly, the output will resemble the following:

```
ETAP configuration [ee:108 se:8 eo:a60 so:acc cw:0 ms:1008008 me:36 mc:20000]
ETAP event table mapped
ETAP subsystem table mapped
ETAP monitored buffer #0 mapped
```

Note: The ETAP configuration values and mapped buffers listed may vary, depending on the selected configuration.

Advanced ETAP Topics

This section is intended for programmers who are faced with the task of writing custom ETAP data collection and analysis utilities. The details presented in this section are subject to change. The advanced topics covered here are:

- Advanced ETAP System Calls
- ETAP File Formats
- Utility and Buffer Interaction
- More about etap_md and monitored buffers
- etap_cget and the cumulative buffer
- Misc. Tid Bits

ETAP File Formats

ETAP Mdata

Each file created by the etap_md daemon contains a single integer header. The integer value equals the number of files created by the etap_md daemon. The etap_md daemon will create one file per processor in the system. For example, if the system contains 4 processors, the etap_md daemon will create 4 files. The header of each file will contain an integer equating to 4.

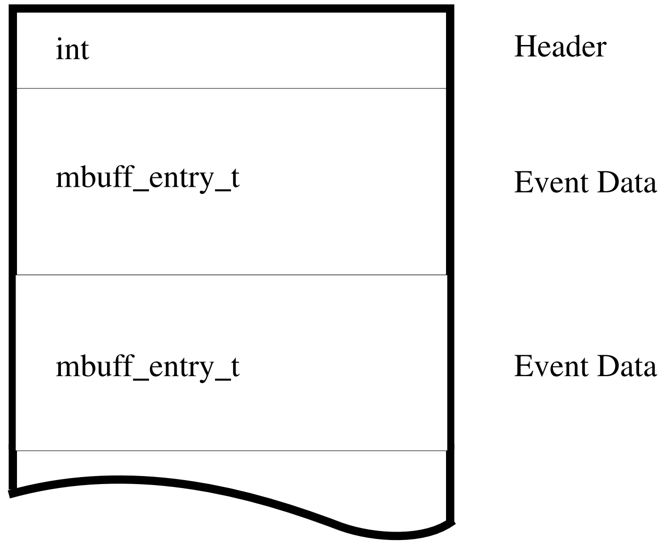


FIGURE 4 **The Mdata file format.**

The mdata header is necessary so that analysis utilities can know how many mdata files they have to deal with. By opening a single mdata file, the analysis utility can learn that it has to open 3 more mdata files for processing.

The rest of the mdata file contains the entries read from the monitored buffer. Each entry is of type mbuff_entry_t.

ETAP Mlost

The etap_md daemon creates a .lost file in Mlost file format. The purpose of the .lost file is to determine how accurately the etap_md daemon collected the data being provided by the kernel. Each processor, monitored by the etap_md daemon, has two integers in the .lost file. The first integer equals the number of monitored buffer entries captured and written to disk. The second integer equals the number of monitored buffer entries lost by the etap_md daemon. The etap_md daemon can loose data if the kernel produces event data faster than the daemon consume it.

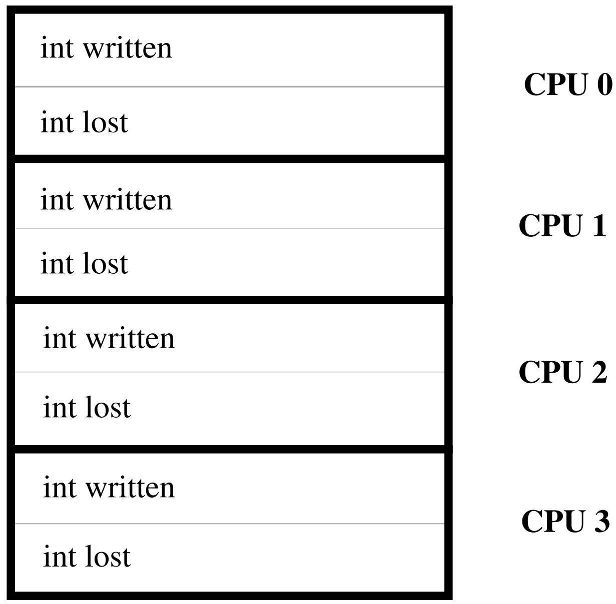


FIGURE 5 **This diagram shows what a 4-processor system Mlost file looks like. Note: A one-processor system would only contain data for CPU 0 (8 bytes, assuming an integer is four bytes long)).**

ETAP Cdata

Each file created by the etap_cstat utility contains a header consisting of 2 integers. The first integer equals the number of cumulative buffer entries recorded in the file. The second integer equals the size of the interval width of the cumulative buffer entries at the time etap_cget was executed.

The rest of the cdata file contains the entries read from the cumulative buffer. Each entry is of type cbuff_entry_t.

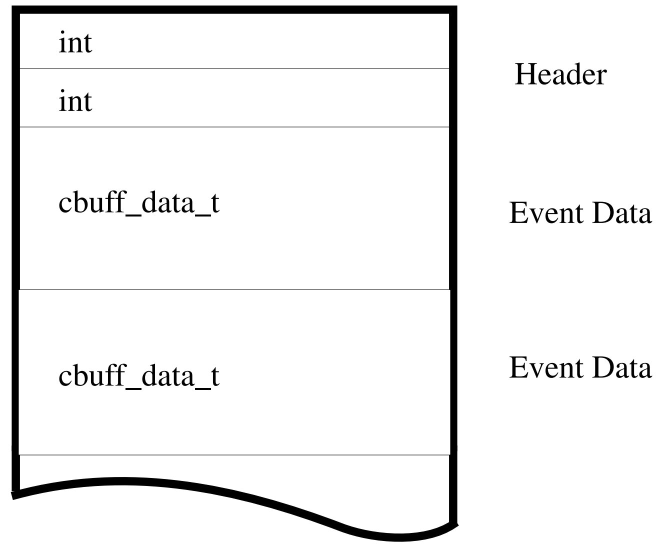


FIGURE 6 **The Cdata file format.**

APPENDIX A

Kernel Event Probes

The kernel has been instrumented with event probes to detect and record specific event occurrences. Event probes (both user and kernel) are allowed to store up to 4 integers of event specific data. This data is optional and depends on the needs of the particular probe. This appendix lists all of the supported ETAP event probes, their function, and the meanings of their event specific data fields.

Note: If the description is blank in the event data table, then the event probe does not make use of the field.

Event Flags

ETAP event probes always record the following information: event type, timestamp, program counter, the ID of the invoking thread, and event flags. This section briefly discusses event flags and how they are used.

Event flags are a set of bit fields which represent operation actions relevant to an event probe. For instance, some probes mark a beginning and an end of an event (e.g., thread life). A single event type is used to atomically toggle trace state for these events, but event flags are used to record the specific event action: begin or end.

The table below lists all of the available event flags. Probes may, and often do, use multiple flags.

Event Flag Table

Event Flag	**Description**
KERNEL_EVENT	The event was recorded by the kernel
USER_EVENT	The event was recorded by a user-level probe
EVENT_BEGIN	The event has begun
EVENT_END	The event has ended
SYSCALL_TRAP	The system call was a Mach trap (not a message)
WRITE_LOCK	A write operation on a complex lock
READ_LOCK	A read operation on a complex lock
COMPLEX_LOCK	A complex lock operation
SPIN_LOCK	A spin lock operation
MUTEX_LOCK	A mutex lock operation

Note: *All flags are set by the kernel. This includes the USER_EVENT flag, which is set automatically by the etap_probe() system call.*

p_act_abort

Function

The p_act_abort event probe records activation abort operations.

Event Data Table

Field	Description
0	ID of aborted activation
1	ID of aborted activation's shuttle
2	
3	

Flags

- KERNEL_EVENT

```
{
        int    id = 0;
        int    count = 0;

        if (argc == 2) max_children = atol(argv[1]);
```

p_interrupt

Function

The p_interrupt event probe traces the entry and exit of interrupt service routines.

Event Data Table

Field	Description
0	Interrupt number
1	
2	
3	

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END

Notes

- The interrupt number is not available at the completion of the interrupt service routine. As a result, the value of the interrupt number is always zero at EVENT_END (see Flags in this Appendix).
- The hardclock interrupt is responsible for updating the real-time clock. The event trace analysis package obtains its timestamps from the real-time clock. As a result, the timestamp for the hardclock interrupt EVENT_BEGIN event is always 10ms behind. The user-level analysis utilities will have to compensate for this timestamp exception.

p_priority

Function

The p_priority event probe traces thread priority changes.

Event Data Table

Field	Description
0	ID of the thread whos priority is effected
1	New priority
2	Old priority
3	

Flags

- KERNEL_EVENT

p_rpc

Function

The p_rpc event probe traces the start and completion of Remote Procedure Call operations.

Event Data Table

Field	Description
0	Destination subsystem
1	Routine number
2	Source task ID
3	Target task ID

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END

Note: The destination subsystem (data field 0) is always equal to zero for RPC return events (EVENT_END).

p_syscall_mach

Function

The p_syscall_mach event probe traces the entry and exit of Mach system calls.

Event Data Table

Field	Description
0	System call number
1	
2	
3	

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END
- SYSCALL_TRAP

p_syscall_unix

Function

The p_syscall_unix event probe traces the entry and exit of UNIX system calls.

Event Data Table

Field	Description
0	System call number
1	
2	
3	

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END

p_thread_ctx

Function

The p_thread_ctx event probe traces the context switches of threads with ETAP trace status activated.

Event Data Table - EVENT_BEGIN

Field	Description
0	ID of thread yielding the processor
1	
2	
3	

Event Data Table - EVENT_END

Field	Description
0	ID of next thread to be scheduled
1	ETAP context switch reason
2	AST quantum reason
3	

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END

p_thread_life

Function

The p_thread_life event probe traces the creation and termination of threads.

Event Data Table

Field	Description
0	Task ID
1	Thread activation ID
2	Thread priority
3	

Flags

- KERNEL_EVENT
- EVENT_BEGIN
- EVENT_END

p_user_events

Function

The p_user_event probe enables user-level probes to record application specific events.

Event Data Table

Field	Description
0	User specified
1	User specified
2	User specified
3	User specified

Flags

- USER_EVENT

Note: All user-levels probes will be ignored if the p_user_event probe is disabled.

APPENDIX A Example Code

This appendix contains an example C language program which demonstrates thread trace control and application instrumentation (user-level probes). The program, etap_ttc, utilizes the rthreads library for multi-threading and the mach library for mach system call definitions.

```
/*
 * File: etap_ttc.c
 *
 * Author: Joe CaraDonna
 *
 * Function: Sample ETAP thread trace control program.
 *           User-level probes are also demonstrated here.
 */

#include <stdio.h>
#include <mach.h>
#include <mach_error.h>
#include <rthreads.h>
#include <mach/etap.h>

#define RECORD_THREAD_ID 0xdeed

unsigned long max_children = 25;

void  child (u_int);

main (int argc, char *argv[])
      /*
       * Spawn specified number of child threads
       */
      while (id < max_children) {
            rthread_spawn((rthread_fn_t) child, (any_t) id);
```

```
		id++;
	}

	/*
	 * Wait for children to complete
	 */
	rthread_wait();

	/*
	 * Terminate task execution
	 */
	exit(0);
}

void child (u_int id)
{
	mach_port_t thread_port;
	kern_return_t ret;
	etap_data_t probe_data;

	/*
	 * Get the thread’s kernel port
	 */
	thread_port = rthread_kernel_port(rthread_self());

	/*
	 * Activate the thread’s ETAP trace status
	 */
	ret = etap_trace_thread(thread_port, TRUE);

	if (ret != KERN_SUCCESS)
		printf (“etap_trace_thread: failed\n”);

	/*
	 * The thread would typically do some work here.
	 *
	 * For kicks, lets insert a user-level probe here to
	 * record the user-level thread ID.
	 *
	 * NOTE:
	 *
	 * Only the first array entry in probe_data is being
	 * used here since we only care about a single integer
	 * value, the thread id.
	 */

	probe_data[0] = id;
	etap_probe( RECORD_THREAD_ID,
		    ETAP_DATA_SIZE,
		    &probe_data);

	/*
	 * Terminate thread execution
	 */
	rthread_exit(0);
}
```