The Event Trace Analysis Package Design Specification

By

Joseph CaraDonna

Revision 1.0 - August 30, 1995

OSF RESEARCH INSTITUTE

11 Cambridge Center
Cambridge, MA 02142

TABLE OF CONTENTS

TABLE OF CONTENTS

1 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.

2 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.

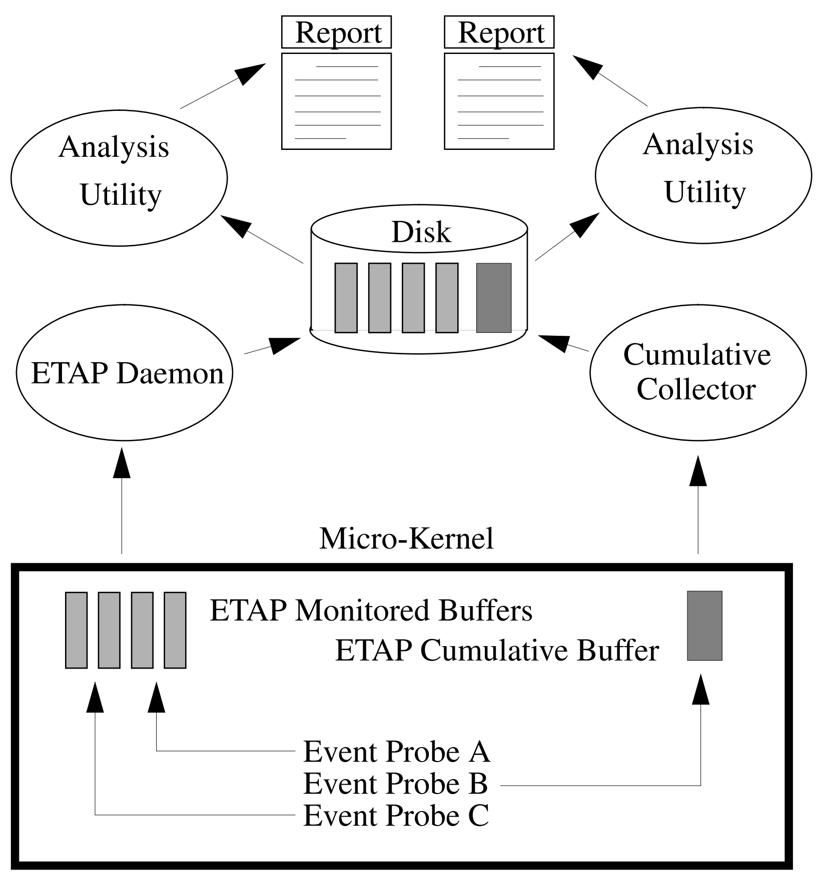


FIGURE 1 **This picture shows the flow of event data from kernel event probes to kernel ETAP buffers to disk. Once the data is written to disk, analysis utilities may generate event reports. Note: four processors are assumed here.**

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.

2.1 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.

2.2 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.

3 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

3.1 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.

3.1.1 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.

3.2 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.

3.2.1 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.

3.2.1.1 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.

3.2.1.2 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.

3.2.1.2.1 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.

3.2.2 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.

3.3 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.

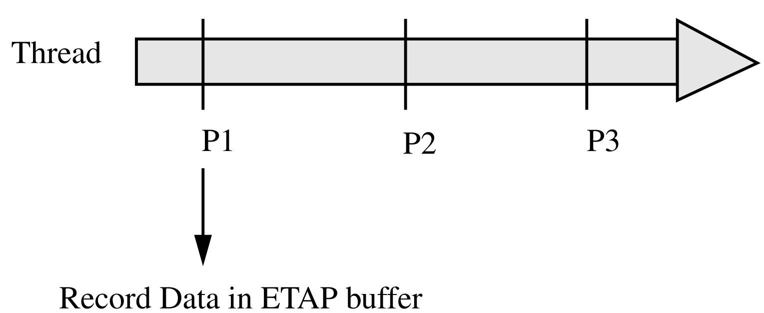


FIGURE 2 **The execution of a thread as it passes kernel ETAP probes.**

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.

4 Kernel Data Structures

The ETAP kernel instrumentation described in the previous sections requires adding fields to existing data structures, as well as, the creation of new structures. This section covers kernel data structure modifications and additions.

4.1 Conditional Build Options

Before data structures can be covered, it is important to understand the ETAP conditional build options. ETAP supports three build options (located in *conf/config.debug*), each of which are mutually exclusive in respect to one another:

1. General event tracing:	**ETAP_EVENT_MONITOR**
2. Monitored lock tracing:	**ETAP_LOCK_MONITOR**
3. Cumulative lock tracing:	**ETAP_LOCK_ACCUMULATE**

To enable the trace package, one ETAP build option is selected (along with the ETAP enable switch). The selected ETAP option determines the level of instrumentation built into the kernel. Option one configures event probes through-out the system. Options two and three add instrumentation to the kernel lock mechanisms.

Further, two conditional definitions, ETAP_LOCK_TRACE and ETAP_MONITOR, are assigned to either one or zero depending on which build option is selected in *conf/config.debug*. The dependencies are as follows:

```
#if      ETAP_LOCK_MONITOR || ETAP_LOCK_ACCUMULATE
#define  ETAP_LOCK_TRACE                     1
#else
#define  ETAP_LOCK_TRACE                     0
#endif

#if      ETAP_LOCK_MONITOR || ETAP_EVENT_MONITOR
#define  ETAP_MONITOR                        1
#else
#define  ETAP_MONITOR                        0
#endif
```

Note: These two build option dependencies are defined in the kern/etap_options.h header file. The kern/etap_options.h header file should be included in all modules which utilize the features of ETAP.

With all that said, the impact of the data structures in the following sections will be easier to understand.

4.2 Data Structure Modifications

The kernel data structures mentioned in this section have been modified to include support for ETAP.

4.2.1 Lock Structures

The *kern/lock.h* header file contains the definitions for the kernel's lock subsystem. To support lock event tracing, fields were added to the various lock data structures.

The **uslock** structure describing machine independent simple (spin) locks and the **mutex_t** structure describing mutually exclusive blocking locks contains the following new fields:

```
#if      ETAP_LOCK_TRACE
         etap_time_t      start_hold_time;     /* Time of last acquisition */
         event_table_t    event_tablep;        /* pointer to event table entry */
#endif
#if      ETAP_LOCK_ACCUMULATE
         cbuff_entry_t    cbuff_entry;         /* cumulative buffer entry */
#endif
#if      ETAP_LOCK_MONITOR
         unsigned int     start_pc;            /* pc where lock operation began */
         unsigned int     end_pc;              /* pc where lock operation ended */
#endif
```

The *start_hold_time* field contains the value of the free running counter (timestamp) when the lock was last acquired. The *event_tablep* field contains the address of a structure that describes the lock event and its status, while the *cbuff_entry* field points to the structure that holds the cumulative data for this lock. Lastly, the *start_pc* and *end_pc* fields contain the program counter of the start and end of the lock event, respectively.

The **lock_t** structure describing complex (multiple reader, single writer) locks contains these new fields:

```
#if     ETAP_LOCK_TRACE
        start_data_node_t    start_list;      /* linked list of start times and pcs */
        event_table_t        event_tablep;    /* pointer to the event table entry */
#endif
#if     ETAP_LOCK_ACCUMULATE
        cbuff_entry_t        cbuff_write;     /* write cumulative buffer entry */
        cbuff_entry_t        cbuff_read;      /* read cumulative buffer entry */
#endif
```

The *start_list* is a pointer to the first start data node in the linked list. The value equates SD_ENTRY_NULL if no start data nodes are present in the list (defined in *kern/etap_pool.h*). The *event_tablep* field contains the address of a structure that describes the lock event and its status. Since complex locks are multiple reader single writer locks, two cumulative buffer entries are required for properly recording event trace data: *cbuff_write* and *cbuff_read*.

4.2.2 Shuttle Structure

The *kern/thread.h* header file contains the definitions which describe the shuttle structure and thread state. Two new fields were added to the shuttle structure:

```
#if     ETAP_EVENT_MONITOR
        int                  etap_reason;     /* real reason why we blocked */
        boolean_t            etap_trace;      /* ETAP trace status */
#endif
```

When a thread context switches, it is desirable to record the reason why the block occurred. The *etap_reason* field is used to pass a reason to the context switch ETAP event probe. Table 1 shows a list of all currently defined blocking reasons (located in *mach/etap_events.h*).

TABLE 1.

Defined Block Reason	Meaning
BLOCKED_ON_UNDEFINED	undefined block
BLOCKED_ON_SEMAPHORE	blocked on a semaphore wait operation
BLOCKED_ON_LOCK	blocked on a lock acquired operation
BLOCKED_ON_LOCK_HANDOFF	blocked on a lock hand-off operation
BLOCKED_ON_MUTEX_LOCK	blocked attempting to acquire a kernel internal mutex lock

TABLE 1.

Defined Block Reason	Meaning
BLOCKED_ON_COMPLEX_LOCK	blocked attempting to acquire a kernel internal reader/ writer lock (complex lock) for writing
BLOCKED_ON_PORT_RCV	blocked waiting to receive a message on a port
BLOCKED_ON_REAPER_DONE	reaper thread finished reaping, blocking itself
BLOCKED_ON_IDLE_DONE	idle thread blocked, as threads are made runnable
BLOCKED_ON_TERMINATION	thread terminated, therefore blocking

Note: More ETAP "blocked on" reasons will be defined as the development of OSF MK continues

The *etap_reason* field is set using the ETAP_SET_REASON macro, defined in *kern/etap_macros.h.* For example, to set the current thread's *etap_reason* field to BLOCKED_ON_LOCK the macro would be called with the following parameters:

```
ETAP_SET_REASON(current_thread(), BLOCKED_ON_LOCK);
```

The *etap_trace* field is used to toggle the ETAP trace status of the thread. When the *etap_trace* field is equal to zero (default), the thread's ETAP events will not be traced. When the *etap_trace* field is equal to one, ETAP events for the thread may be recorded. Note: Control of this toggle bit is provided to application programmers via the **etap_trace_thread()** system call.

Note: Shuttles are traced rather than activations. This scheme allows us to trace a thread as it migrates from task to task.

4.3 New Data Structures

The kernel data structures mentioned in this section have been added in order to support ETAP. All of the new ETAP data structures are defined in *mach/etap.h*.

4.3.1 Complex Lock Start Lists

Unlike simple locks and mutex locks, each complex lock maintains a linked list of start data. Start data is information gathered at the beginning of a lock event (when tracing for that lock is enabled), such as: *start_wait_time, start_hold_time*, and *start_pc*. Using the kernel's start data service, **get_start_data_node()**, a node is allocated and assigned to each thread currently performing a complex lock operation. Even though only one node will be in the start_list when a write operation is in progress, several may exist when simultaneous read operations are executing. This scheme is necessary to prevent multiple readers from trashing the start_data. A start_list node is defined in *kern/etap_pool.h as*:

```
struct start_data_node {
        unsigned int            thread_id;          /* thread id */
        etap_time_t             start_hold_time;    /* time of last acquisition */
        etap_time_t             start_wait_time;    /* time of first miss */
        unsigned int            start_pc;           /* pc of acquiring function */
        unsigned int            end_pc;             /* pc of relinquishing function */
        struct start_data_node *next;               /* pointer to next list entry */
};

typedef struct start_data_node* start_data_node_t;
```

4.3.2 Cumulative Buffer Entries

The cumulative buffer, defined in the next section, is primarily made-up of cbuff_entry structures. The cbuff_entry structure stores cumulative information for a particular lock event (ETAP_LOCK_ACCUMULATE configuration only). When a simple, or mutex lock, is initialized for the first time, a cbuff_entry structure is reserved in the cumulative buffer and the address of the reserved entry is stored in the associated lock's cbuff_entry pointer variable (see Data Structure Modifications). If a lock is re-initialized, then its cbuff_entry structure is also re-initialized.

Unlike simple and mutex locks, complex locks have two cumulative buffer pointers, in order to accumulate data for read and write operations separately. A cumulative buffer entry is reserved and assigned to cbuff_write upon lock initialization (like *cbuff_entry* for simple locks). An entry for *cbuff_read*, however, is only reserved and initialized if a read operation is performed, otherwise the *cbuff_read* pointer is assigned CBUFF_ENTRY_NULL. The *cbuff_entry* structure is defined in *mach/etap.h* as follows:

```
struct cbuff_entry {
        etap_event_t        event;              /* event type */
        u_short             kind;               /* read, write, simple or mutex */
        u_int               instance;           /* & of event struct */
        struct cbuff_data   hold;               /* hold trace data */
        struct cbuff_data   wait;               /* wait trace data */
        u_long              hold_interval[ETAP_CBUFF_IBUCKETS];    /* hold interval array */
        u_long              wait_interval[ETAP_CBUFF_IBUCKETS];    /* wait interval array */
};

typedef struct cbuff_entry* cbuff_entry_t;
```

The event field identifies the type of lock being recorded in the buffer entry (e.g., thread lock, queue lock, VM object lock, etc.). The kind field indicates the kind of lock: simple, mutex or complex. The instance field stores the address of the lock being traced. If the lock is dynamic, then the instance field is NULL (see Dynamic Locks below). The lock event statistics are stored in the hold and wait *cbuff_data* structures, which are defined as follows:

```
struct cbuff_data {
        u_long          triggered;      /* number of event occurrences */
        etap_time_t     time;           /* sum of event durations   */
        etap_time_t     time_sq;        /* sum of squared durations*/
        etap_time_t     min_time;       /* min duration of event */
        etap_time_t     max_time;       /* max duration of event    */
};
```

4.3.3 Monitored Buffer Entries

The ETAP monitored buffers primarily consist of mbuff_entry structures. The mbuff_entry structure describes a single ETAP event and is defined as follows:

```
struct mbuff_entry {
        u_short         event;          /* event type */
        u_short         flags;          /* event strain flags */
        u_int           instance;       /* address of event (lock, thread, etc.) */
        u_int           pc;             /* program counter */
        etap_time_t     time;           /* operation time */
        etap_data_t     data;           /* event specific data */
};

typedef struct mbuff_entry* mbuff_entry_t;
```

The event field identifies the ETAP event, while the flag field indicates the type of operation being performed (e.g., marks the beginning or end of an event). Table 2 shows all of the supported flag values.

TABLE 2. Flag field bit field values

FLAG	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CUM_CONTENTION																X
CUM_DURATION															X	
MON_CONTENTION														X		
MON_DURATION													X			
WRITE_LOCK												X				
READ_LOCK											X					
COMPLEX_LOCK											X	X				
SPIN_LOCK										X						
MUTEX_LOCK									X							
USER_EVENT								X								
KERNEL_EVENT							X									
EVENT_END						X										
EVENT_BEGIN					X											
SYSCALL_TRAP				X												
reserved	X	X	X													

Note: Each flag value is a bit field. Hence, It is common for multiple bits to be set simultaneously.

The instance field contains the address of the lock event or thread invoking the event. If lock event tracing is configured, then the instance contains the address of the lock being operated on. If thread event tracing is configured the instance contains the address of the thread invoking the event. The pc field contains the program counter at the time of the event occurrence. The time field stores the timestamp of the event. Lastly, the data field is an array of unsigned integers used to store event specific data:

```
typedef unsigned int    etap_data_t[ETAP_MBUFF_DATASIZE];
```

The value of ETAP_MBUFF_DATASIZE is typically equal to four, but this may be configured at compile time.

4.3.4 Cumulative Lock Event Buffer

Cumulative lock event entries are stored in a global cumulative buffer, defined by the cumulative_buffer structure shown below.

```
struct cumulative_buffer {
        u_long              next;           /* next available entry in buffer */
        u_short             static_start;   /* first static entry in buffer */
        struct cbuff_entry  entry [ETAP_CBUFF_ENTRIES];     /* buffer entries */
};
```

ETAP_CBUFF_ENTRIES is defined to be the maximum number of entries the cumulative buffer can store (configurable at compile time). The *next* header variable indicates the next available entry in the buffer. Each time a cumulative buffer entry is reserved, the *next* variable is incremented by one (after the entry was reserved). The *static_start* variable is assigned to the entry where static lock entry reservations begin (this is made clearer in the Dynamic Locks section).

The cumulative buffer is protected by a spin lock (*cbuff_lock*) which must be acquired before an entry is reserved. However, the actual entries, themselves, are protected by the particular lock being traced. Thus, no additional locking is required to increment fields in the entries while the traced lock is being held.

The actual size of the cumulative buffer is rounded up to a page multiple so that it can be mapped by the server and user-tasks. Once the buffer is full, subsequent entry reservation requests fail, returning CBUFF_ENTRY_NULL.

4.3.5 Dynamic Locks

Each type of dynamically allocated lock (e.g., thread locks) has an associated cumulative entry. These entries are reserved at the head of the cumulative buffer and their offsets are well-known. The *static_start* cumulative buffer header variable marks the boundary between the dynamic lock entries and the static lock entries. Static lock entry reservations begin at the *static_start* offset.

Data for dynamic locks is maintained on a per-lock-type basis. Each dynamic lock entry is protected by its own spin lock.

4.3.6 Monitored Buffers

General event or monitored lock tracing modes record events in the monitored buffer(s). ETAP dynamically allocates one monitored buffer per-processor. Each monitored buffer is defined by the following data structure:

```
struct monitor_buffer {
        u_long               free;          /* index of next available record */
        u_long               timestamp;     /* timestamp of last wrap around */
        struct mbuff_entry   entry[1];      /* buffer entries (place holder) */
};

typedef struct monitor_buffer* monitor_buffer_t;
```

Note: All monitored buffers are circular

Each buffer consists of an unsigned long describing the next available buffer entry (*free*), a timestamp incremented when the buffer wraps around, and a configurable number of event entries. Monitored buffers are rounded to the next page boundary. No locking is required because entries are assigned on demand and there is no contention. However, interrupts must be disabled while the next *free* entry index is being obtained.

Upon kernel initialization, the address of each monitored buffer is stored in the *mbuff* array (declared and initialized in *kern/etap.c*). The appropriate monitor buffer is located at run-time by using the current processor number to index the *mbuff* array.

4.3.7 Subsystems and Event Types

The kernel implements two ETAP tables for event identification and classification purposes: event table and the subsystem table.

The event table contains a database of all of the system events and their current trace status. Each event is described by the *event_table_entry* structure (*mach/etap.h*):

```
struct event_table_entry {
        u_short      event;                          /* etap event type */
        u_short      status;                         /* event trace status */
        char         name [EVENT_NAME_LENGTH];       /* event text name */
        u_short      dynamic;                        /* dynamic ID (0=none) */
};
```

The *event* field is used to store a unique event type (a number used to identify a specific event). The *status* field contains the current trace status of the event (i.e., is the event probe on or not). The *name* field holds an ASCII text name for the event, which is used as a handle. Lastly, the *dynamic* field indicates whether the event is a dynamically allocated lock or not. If the event does not represent a dynamic lock, then the value is assigned to zero. Otherwise, the *dynamic* variable is assigned a unique number, which is used to index the cumulative buffer (when cumulative lock tracing is configured).

Note: All of the fields of the event table entries are read-only except for the status field, which can be read or written-to from the user-level (via the etap_trace_event system call).

The subsystem table classifies related events into subsystems (e.g., all of the lock event types related to virtual memory are in the ETAP_SUBS_LOCK_VM subsystem). Each subsystem is described by the *subs_table_entry* structure (*mach/etap.h*):

```
struct subs_table_entry {
        u_short         subs;                           /* etap subsystem type */
        char            name [EVENT_NAME_LENGTH];       /* subsystem text name */
};
```

The *subs* field stores a unique subsystem number, while the *name* field contains an ASCII text name for the subsystem.

Note: Lists of all supported event types and subsystems are located in the mach/etap_events.h header file. The event table and subsystem table are statically allocated and initialized in the kern/etap.c module. If a new event or subsystem is added to a list in etap_events.h, then it must also be added to the appropriate table in etap.c.

4.3.8 Time

ETAP introduces a new time format for internal use, as well as, export to the ETAP user-level utilities: *etap_time*. The reason *etap_time* is used instead of some already existing format (like tvalspec) is that it allows for simple kernel-internal timestamp manipulation. At times, ETAP performs difference, summation and comparison operations of 64bit timestamps. These calculations must be performed quickly and efficiently. The *etap_time* structure is defined in *mach/etap.h* as follows:

```
union etap_time {
        double          value;
        struct {
                        u_int           lo;
                        u_int           hi;
        } stamp;
};
typedef union etap_time etap_time_t;
```

Note: The etap_time data structure is machine dependent (due to byte ordering and clock support reasons) and should be defined in a machine dependent header file.

Notice that the *etap_time* structure is a union of variables *value* and *lo/hi*. As one might guess, *lo* stores the low-order 32bits of the timestamp and *hi* stores the hi-order 32bits. The *value* variable is used to address the entire 64bit timestamp. The kernel deals with the two 32bit values, while the user-level utilities use the 64bit *value*.

Note: The user-level analysis utilities are responsible for converting the two 32bit values into a single float: value.

The following portion of C language code demonstrates how a user-level utility should perform the timestamp format conversion:

```
/*
* Convert raw 64bit nanosecond clock time to
* microsecond float format
*/

double                          temp_time;
struct etap_time                time;

temp_time = (double) time.stamp.hi * pow(2, 32);
temp_time = temp_time + (double) time.stamp.lo;
time.value = temp_time / 1000;
```

5 ETAP Initialization

The ETAP buffers described in previous sections are initialized during start-up and mapped into the OSF/1 Single Server's address space, as detailed in the following sub-sections.

5.1 Cumulative Buffer

The cumulative buffer is used to store aggregate lock performance statistics for kernel lock events (when cumulative lock tracing is configured). Each lock reserves an entry (or *entries* in the case of complex locks) from the buffer as they are initialized. Hence, the cumulative buffer must be allocated and initialized prior to the first lock initialization. The cumulative buffer is statically allocated in the *kern/etap.c* module, since locks begin to initialize themselves long before dynamic memory allocation services are available (e.g., kmem_alloc). The cumulative buffer's initialization routine, *etap_init_phase1()*, is also called prior to the first lock initialization.

5.1.1 Page Alignment

Mach only supports memory mapping operations on a per-page basis. Hence, the cumulative buffer must be page aligned to allow user-level utilities to map the buffer into their address space. As mentioned above, the cumulative buffer is statically allocated. The problem is that the C language does not support primitives to force static data regions to be page aligned. The solution lies dynamically aligning the buffer at run-time.

The cumulative buffer is statically allocated to be larger than we need (by exactly two pages). Upon buffer initialization, the cumulative buffer pointer *(cbuff)* is assigned to the statically allocated buffer region (*cbuff_allocated*), using the following equation:

```
cbuff = (cumulative_buffer_t) round_page (&cbuff_allocated);
```

The equation rounds the address of the buffer up to the nearest page boundary, aligning it, and preparing it for memory mapping. The second *extra* page is allocated for security purposes, preventing curious users from accessing kernel internal data.

5.1.2 Configurable Options

The cumulative buffer supports compile-time and run-time configuration options. At compile-time, the following parameters may be configured in *mach/etap.h*:

- The number of entries in the buffer: ETAP_CBUFF_ENTRIES
- The number of interval buckets: ETAP_CBUFF_IBUCKETS
- The initial interval width: ETAP_CBUFF_WIDTH

At run-time, the interval width may be modified using the *etap_trace_event()* system call.

5.2 Monitored Buffers

The monitored buffers are used to store either general event trace data from tracing thread events or monitored lock trace data. One monitored buffer is allocated per-processor. Unlike the cumulative buffer, monitored buffers do not have to be allocated prior to the availability of dynamic memory allocation services. In fact, monitored buffers are dynamically allocated via the *kmem_alloc()* routine in *etap_init_phase2()*. A benefit of dynamically allocating memory is that the allocated region is page aligned, so we don’t have to perform the alignment tricks practised by the cumulative buffer.

5.2.1 Configurable Options

The monitored buffers support compile-time configuration options. The following parameters may be configured in *mach/etap.h*:

- The number of entries available in a monitored buffer: ETAP_MBUFF_ENTRIES
- The size of the event specific data region: ETAP_MBUFF_DATASIZE

5.3 Mapping ETAP Buffers and Tables

As already mentioned, the kernel's ETAP buffers are allocated either statically or during the kernel's bootstrap process. However, they must eventually be mapped into the address space of the task performing the data collection. In order to accomplish this, the OSF/1 Single Server is used as an intermediary. The buffers are first mapped into the server's address space (upon boot-up) and later into a client task's address space.

This dual mapping method is accomplished by creating a pseudo-device representing the ETAP buffers and tables. Each buffer or table is distinguished by utilizing the minor device numbers of the driver, as follows:

- Event table: ETAP_TABLE_EVENT (0)
- Subsystem table: ETAP_TABLE_SUBSYSTEM (1)
- Cumulative buffer: ETAP_BUFFER_CUMULATIVE (2)
- Monitored buffers: ETAP_BUFFER_MONITORED+ CPU number (3 to ?)

Since there may be multiple monitored buffers, multiple minor device numbers are required to represent each buffer. The reserved monitored buffer minor device numbers range from 3 to 2+number of CPUs. For example, if your system contains four processors then the minor device numbers representing the monitored buffers range from 3 to 6 (where 3 represents CPU zero's monitored buffer and 6 represents CPU four's buffer).

The server opens the ETAP pseudo-device and maps the buffers and tables it represents into its address space. Upon receiving requests from user tasks, the server re-maps the buffers and tables into the clients address space.

Before the server can map the buffers into its address space, it must first acquire information about the buffer and table configurations via the *etap_get_info()* system call. The system call returns the size of each buffer and table (for more information see Section-?). Once the buffer information is obtained, the mapping sequence (for each buffer and table) may commence as follows:

- **Open the device**: The server calls *device_open*() on the name "etap_mapX", where X is the minor device number of the requested buffer or table. Opening a device causes the port representing the device to be returned.
- **Map the device**: Using the newly acquired device port, the device representing the buffer (or table) may be mapped via *device_map*(). The *device_map()* system call returns the device's pager port.
- **Map the memory region**: Once the device has been opened and mapped, the memory region it represents may be mapped into the servers address space via the *vm_map()* system call. The *vm_map()* system call creates a memory object, initializes its pager, and maps the memory region into the specified address space (that of the server).

Now that the server has all of the ETAP buffers and tables mapped, client tasks may request mappings via the Server's *etap_map()* system call (see Section-? for details).

An easier scheme to map the ETAP buffers and tables into the clients address space may exist, but the method described above is already employed by the microkernel and server to map kernel structures into a clients address space. Thus, it is easier to reuse the mechanisms than invent new ones.

6 ETAP System Calls

Some of the functionality described in this specification is implemented via new system calls to both Mach and the OSF/1 Single Server. This section provides functional descriptions and parameter synopses for each of the following system calls: *etap_probe, etap_trace_thread, etap_trace_event, etap_get_info, etap_map, etap_info*.

6.1 etap_probe

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

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

The *event_id* parameter is used to pass a user defined value which can be used for event identification. The *data_size* parameter describes the size (in bytes) of the data being passed. The *data* parameter is a pointer to the event data being passed to the kernel for logging.

Note: The data size is limited to ETAP_DATA_SIZE, which is defined in mach/etap.h.

6.2 etap_trace_thread

The *etap_trace_thread()* Mach system call (defined in libmach) is used to toggle the ETAP trace status of a thread. The system calls synopsis is as follows:

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

The *target_thread* parameter contains the port of the thread who's ETAP trace status will be toggled. The *active* parameter is a boolean value (either TRUE or FALSE) which states 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.

6.3 etap_trace_event

The *etap_trace_event()* Mach system call (defined in libmach) is used to enable and disable kernel event probes and all modes of lock event tracing. The call also supports a reset option, where the cumulative buffer data and all event type tracing is reset to zero. When the reset option is used, a new interval width can also be defined, using the *nargs* parameter. Its synopsis is as follows:

```
kern_return_t etap_trace_event
        (unsigned short        mode,
        unsigned short         type,
        boolean_t              enable,
        unsigned int           nargs,
        unsigned short         *args);
```

The *mode* parameter indicates the desired trace flavor or reset option and may be one, or more, of the following:

- ETAP_CUMULATIVE: cumulative lock event trace mode
- ETAP_MONITORED: monitored event trace mode (for probes or locks)
- ETAP_RESET: reset mode, clears all trace status and cumulative buffer entries

The *type* parameter is used when measuring lock event contention or durations and may be one, or more, of the following:

- ETAP_CONTENTION
- ETAP_DURATION

The *enable* parameter is a boolean value which indicates whether the event trace operation is to be enabled (TRUE) or disabled (FALSE). The *nargs* parameter specifies how many arguments are passed in the *args* array. Each element of the *args* array is a character string representing a specific subsystem or event type. These values must correspond to the ones the kernel uses to represent the same subsystems and event types. The maximum length of a character string is EVENT_NAME_LENGTH (defined in *mach/etap.h*).

To reset the ETAP instrumentation, the system call would utilize the *mode* parameter, passing the value of ETAP_RESET (All other parameters may equal NULL). If, at the time of reset, the *nargs* parameter is assigned a value, then the cumulative buffer interval width will be adjusted to be the size of that value.

For example, the following system call would reset the ETAP instrumentation and adjust the cumulative buffer's interval width to 100ms:

```
etap_trace_event (ETAP_RESET, 0, 0, 100, 0);
```

6.4 etap_get_info

The Server executes the privileged *etap_get_info()* Mach system call to determine the configuration of ETAP within the microkernel. The synopsis is as follows:

```
kern_return_t etap_get_info
        (mach_port_t        priv_port,
         int                *et_entries,
         int                *st_entries,
         vm_offset_t        *et_offset,
         vm_offset_t        *st_offset,
         int                *cb_width,
         int                *mb_size,
         int                *mb_entries,
         int                *mb_cpus)
```

The *priv_port* parameter is the name of the Server's privileged device port (server_device_port), granting the Server access to this service. The *et_entries* and *st_entries* arguments return the number of entries in the event and subsystem tables, respectively. The *et_offset* and *st_offset* arguments return the page offsets of the event and subsystem tables, respectively. The *cb_width* parameter returns the current cumulative buffer interval width. The *mb_size* parameter returns the size of the monitored buffers, while the *mb_entries* argument returns the maximum number of entries in a monitored buffer. Lastly, the *mb_cpus* field returns the number of allocated monitored buffers (or supported CPUs).

The Server calls *etap_get_info()* at boot-time in order to properly map the ETAP buffers and tables into its address space. When mapping is successfully completed, the Server displays the ETAP configuration on the console. The configuration output will resemble the following:

```
ETAP configuration [ee:108 se:8 eo:a60 so:acc cw:0 ms:1008008 me:36 mc:1]
ETAP event table mapped
ETAP subsystem table mapped
ETAP monitored buffer #0 mapped
```

The values between the brackets correspond with the arguments returned by the *etap_get_info()* call (listed above). For example, "ee:108" indicates that the event table contains 108 ETAP event types and "mc:1" indicates that one monitored buffer has been allocated.

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

6.5 etap_map

The *etap_map()* sytem call, handled by the OSF/1 Server, enables applications to map the ETAP buffers and tables into their address space for reading. The declaration of the call is as follows:

```
int etap_map
        (int            type,
         int            cpu,
        vm_address_t    *addr)
```

The *type* parameter indicates the type of ETAP buffer or table to be mapped, and may be one of:

- ETAP_TABLE_EVENT, for the event table
- ETAP_TABLE_SUBSYSTEM, for the subsystem table
- ETAP_BUFFER_MONITORED, for a monitored buffer
- ETAP_BUFFER_CUMULATIVE, for the cumulative buffer

The *cpu* parameter is used to select a specific monitored buffer and is only valid when ETAP_BUFFER_MONITORED is selected for the type. The address in the user task, where the buffer is mapped, is returned in the *addr* parameter.

The system call returns zero on success, and a non-zero value upon failure.

6.6 etap_info

When the OSF/1 Server receives an *etap_info()* system call, it returns some of the configuration data it received from the *etap_get_info()* Mach system call. The synopsis is as follows:

```
int etap_info
        (int     *event_table_entriesp,
        int      *subs_table_entriesp,
        int      *cbuff_widthp,
        int      *mbuff_sizep,
        int      *mbuff_entriesp,
        int      *mbuff_cpusp)
```

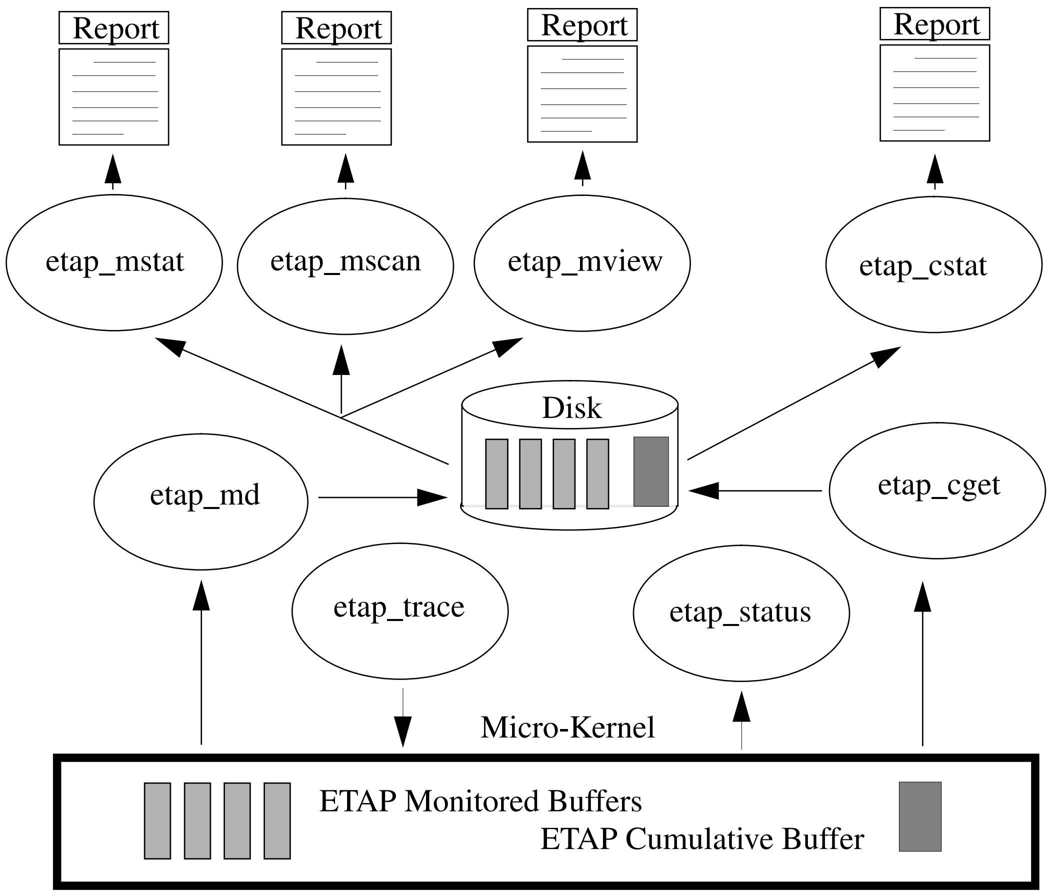
The *event_table_entriesp* and *subs_table_entriesp* arguments return the number of entries in the event and subsystem tables, respectively. The *cbuff_widthp* parameter returns the current cumulative buffer interval width. The *mbuff_sizep* parameter returns the size of the monitored buffers, while the *mbuff_entries*p argument returns the maximum number of entries in a monitored buffer. Lastly, the *mbuff_cpusp* field returns the number of allocated monitored buffers (or supported CPUs).

Note: Unlike the etap_get_info() Mach system call, table offsets are not returned to the user task for security reasons.

The system call returns zero on success, and a non-zero value upon failure.

7 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.



7.1 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

7.2 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.

7.2.1 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 lock_vm

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

% etap_trace -R -i 120

7.3 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.

7.4 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* is used. 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.

7.5 etap_mscan - Event 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.

Note: The etap_mscan tool is a simple program, from which more complex tools can be derived to manage and give meaning to the traced data.

7.6 etap_mview - Event Analysis Utility

The *etap_mview* utility is used to view the contents of Mdata and Mlost files created by the ETAP daemon (*etap_md*). This utility is an enhanced version of *etap_mscan*, adding several features for better understanding of the collected data. A synopsis is as follows:

etap_mview [-i filename] [-v] [-p]

-i <filename> Specify a base file name other than the default /tmp/etap_mdata.log.

-v Indicate the validity of the statistics being reported. Specifically note the number of records written, the number of records lost, and calculate an accuracy percentage for each processor.

-p Display system tables: exception names and numbers, interrupt names and numbers, Mach system call names and their associated system call numbers, Mach message names and their associated message IDs, UNIX system call names and their associated system call numbers.

The biggest difference between the *etap_mscan* and the *etap_mview* utilities is that *etap_mscan* simply provides a raw data dump of Mdata file contents, while *etap_mview* displays ETAP events in more meaningful, and therefore helpful, manner.

Note: The etap_mview analysis utility is the preferred tool for viewing Mdata files.

7.7 etap_mstat - Lock Analysis Utility

The *etap_mstat* utility is used to analyze monitored lock event data. Hence, *etap_mstat* generates statistical reports from files created by the *etap_md* daemon (see Section-?). The *etap_mstat* command offers various options for reporting monitored trace data. A synopsis follows:

etap_mstat [-t] [-a] [-c] [-e type] [-s subsystem] [-h] [-w] [-v] [-r] [-d] [-b #] [-u time]
[-g time] [-i input]

These options allow for the same data to be reported in several different ways, as described below:

-t Provide a terse statistical summary of the lock data, including for each lock selected: the mean, standard deviation, variance, as well as calculated minimum and maximum times. The type of operation is also identified: simple, mutex, read or write.

-a Provide the same information as the -t option, but offers aggregate information on a per lock type basis instead of per individual lock record. The kind of lock is also identified: simple, mutex or complex.

-c Provide the cumulative distribution information.

-e <type> Select records of a particular lock event type. Several lock event types may be selected by preceding each one with the -e option.

-s <subsystem> Select records of a particular subsystem. Several subsystems may be selected by preceding each one with the -s option.

-h Report lock hold times in msec.

-w Report time spent waiting to acquire contended locks in msec.

-v Indicate the validity of the statistics being reported. Specifically note the number of records written, the number of records lost, and calculate an accuracy percentage for each processor.

-r Print the names of the functions that acquired and released the lock.

-d Display verbose output (current configuration and actions).

-b <#> Change the size of the file buffer (default is one megabyte).

-u <time> Select records with measured times less than or equal to time msec.

-g <time> Select records with measured times greater than or equal to time msec. A range is specified by -g time1 -u time2.

-i <file> Provide an alternate input file containing traced monitored lock data. If this option is omitted, /tmp/etap_mdata.log is used.

The default output produced by *etap_mstat* is the long form, printing each lock record selected, as well as providing the statistical per lock information available with the -t option and the per lock event type information offered by the -a option. The default output also includes the validity information supplied by the -v option, computed from the corresponding .lost file. Also, the program counters printed are correlated to the kernel's symbol table for easier identification of functions that require modification.

The *etap_mstat* tool carries out object conversion, record storing, and processing in the same way as *etap_cstat*. Each selected lock's statistical information is displayed in the requested format.

7.7.1 Usage

To demonstrate the flexibility of the etap_mstat utility, examples of its usage are supplied below.

Note: The examples assume that the tracing and obtaining of data has already been executed via the etap_trace and etap_md utilities.

Display the validity report showing the data lost/recorded ratio:

```
% etap_mstat -v

cpu     written  lost    validity
-----------------------------------------------------------
0       102250   21      99.9%
1       100000   0       100.0%
2       130000   300     99.7%
3       110010   3050    97.3 %
```

Display the aggregate hold distribution of the ipc_port_ip lock with bucket widths of 20:

```
% etap_mstat -a -h -c 20 -e ipc_port_ip

ipc_port_ip     1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE       mean    stdv    min     max     accesses        kind
-----------------------------------------------------------------------------------------------
hold            157     187     16      1356    108             simple
-----------------------------------------------------------------------------------------------
```

A-I	0 - 19	20 - 39	40 - 59	60 - 79	80 - 99	100 - 119	120 - 139	140 - 159	160 - 179	180 -
hold	35	4	21	0	0	0	2	2	0	46

Display the names of all the functions who held the ipc_port_ip lock for longer then one millisecond:

```
% etap_mstat -r -h -e ipc_port_ip -g 1000

ipc_port_ip     2 type instances
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
HOLD            time    op      offset  function name
-------------------------------------------------------------------------------------
locker          1356    simple  160     _ipc_right_clean
unlocker                        196     _ipc_right_destroy
-------------------------------------------------------------------------------------
locker          1121    simple  160     _ipc_right_clean
unlocker                        196     _ipc_right_destroy
-------------------------------------------------------------------------------------
```

7.8 etap_cget

The kernel collects cumulative lock event data in the ETAP cumulative buffer. Because this buffer is allocated and maintained in the kernel, a utility is needed to provide user-access to this data. Hence, the etap_cget utility writes lock trace data from the kernel's cumulative buffer to a file on disk. A synopsis of its usage follows:

etap_cget [-o file]

Users may specify an output file via the -o option; otherwise */tmp/etap_cdata.log* is used. When this command is executed, it invokes the *etap_map* system call described in Section-?, passing it the ETAP_BUFFER_CUMULATIVE option. This system call maps the kernel's cumulative lock buffer into the address space of the *etap_cget* task. Once the mapping is complete, the utility simply writes all of the data from the cumulative lock buffer to disk and exits.

7.9 etap_cstat - Lock Analysis Utility

The etap_cstat utility is used to analyze cumulative lock event data. Hence, etap_cstat generates statistical reports from files created by the etap_cget utility. The etap_cstat command offers various options for reporting cumulative lock data. A synopsis follows:

etap_cstat [-t] [-a] [-c] [-l type] [-s subsystem] [-h] [-w] [-i input]

These options allow the same data to reported in several different ways, as described below:

-t Provide terse output containing a statistical summary of the cumulative data, including for each record selected: the mean, standard deviation, and variance, as well as the minimum and maximum times recorded per lock. The type of lock operation is also identified: simple, read or write.

-a Provide the same information as the -t option, but offers aggregate information on a per lock type basis instead of per individual lock record. The kind of lock is identified: simple, complex or mutex.

-c Provides cumulative interval information only without statistical analysis.

-e <type> Selects records of a particular lock event type. Several lock event types may be selected by preceding each one with the -e option.

-s <subsystem> Select records of a particular subsystem. Several subsystems may be selected by preceding each one with the -s option.

-h Report lock hold times in microseconds.

-w Report time spent waiting to acquire contended locks in microseconds.

-i <file> Provide an alternate input file containing lock data. If this option is omitted, /tmp/etap_cdata.log is used.

The default output produced by *etap_cstat* is the long form, providing a record per each lock entry containing the statistical information provided by the -t option and the cumulative interval information offered by the -c option, as well as the aggregate summary information per lock type given with the -a option.

The *etap_cstat* tool first converts the objects, specified by the -e or -s options, from their user-format (text-name) to their kernel format (event type or subsystem ID). The conversion is performed by mapping the appropriate kernel table into the task's address space (using the *etap_map()* system call) and performing look-ups. A look-up consists of locating the text-name in the table, cross referencing its associated event type and storing the event type in an object list. If the -s or -e options are not selected, the object list contains all of the event types.

Once the object list is created, *etap_cstat* opens the appropriate input file, maps the kernel's cumulative buffer and continuously stores all records whose event type is listed in the object list. Each stored record is categorized, within a multidimensional, probabilistic skip list data structure, by subsystem, event type, and instance for efficient processing.

On completion of record collection, data processing begins. Processing the records consists of traversing the skip lists, performing any statistical calculations and displaying the information in the format requested (default long form or otherwise).

7.9.1 Usage

To demonstrate the flexibility of the etap_cstat lock analysis utility, examples of its usage are supplied below.

Note: The examples assume that the tracing and obtaining of data has already been executed via the etap_trace and etap_cget utilities.

Display aggregate and distribution data for the ipc_object lock event type:

```
% etap_cstat -a -c -e ipc_object

ipc_object                1 event type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max      accesses        kind
------------------------------------------------------------------------------------------------
hold         291     17      12     58840    233945          simple
wait         491     447     16     21068    10752
```

A-I	0 - 59	60 - 119	120 - 179	180 - 239	240 - 299	300 - 359	360 - 419	420 - 479	480 - 539	540 -
hold	121442	56379	19968	5029	1052	848	755	648	503	2731
wait	2141	4685	942	340	171	118	90	61	73	2131

Display only aggregate hold data for the network subsystem:

```
% etap_cstat -a -h -s net

net_se                  1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max    accesses    kind
------------------------------------------------------------------------------------------
hold         27      4       20     32     14          simple
------------------------------------------------------------------------------------------

net_qfree               1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max    accesses    kind
------------------------------------------------------------------------------------------
hold         45      6       40     52     4           simple
------------------------------------------------------------------------------------------

net_q                   1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max    accesses    kind
------------------------------------------------------------------------------------------
hold         36      7       28     48     6           simple
------------------------------------------------------------------------------------------
```

Display only aggregate wait data for the zone and vm_object locks from the /tmp/dizzy.cd cumulative data file:

```
% etap_cstat -a -h -e zone -e vm_object -i /tmp/dizzy.cd

zone                    1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max      accesses    kind
------------------------------------------------------------------------------------------
wait         192     361     16     1500     92          simple
------------------------------------------------------------------------------------------

vm_object    1 type instance
*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
AGGREGATE    mean    stdv    min    max      accesses    kind
------------------------------------------------------------------------------------------
wait         2510    536     20     16712    1297        simple
------------------------------------------------------------------------------------------
```

8 ETAP Configuration

While ETAP configuration options are already scattered through-out this document, this section focuses on the need-to-know compile-time configuration options necessary to build a default ETAP system. The event trace analysis package requires both the kernel and the server to be configured with ETAP options enabled.

8.1 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.

8.2 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:1]
ETAP event table mapped
ETAP subsystem table mapped
ETAP monitored buffer #0 mapped

9 ETAP File Formats

The ETAP utilities, etap_md and etap_cget, collect event data from the kernel and write the information to files. This section covers the file formats used by these utilities, so that analysis utilities (e.g., etap_mview and etap_cstat) can be written to process the files properly.

9.1 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.

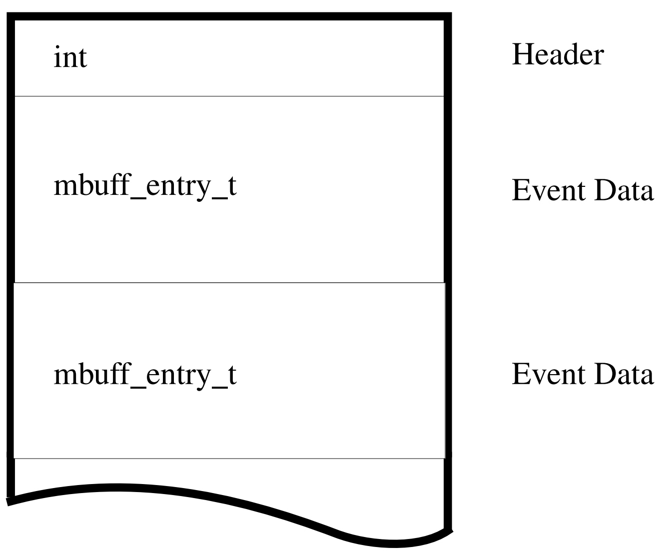


FIGURE 3 **The Mdata file format.**

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*.

9.2 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.

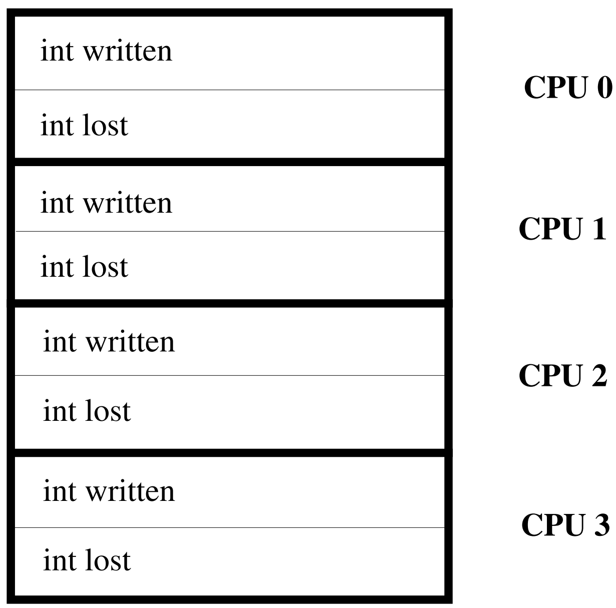


FIGURE 4 **This diagram shows what a 4-processor system Mlost file looks like. Note: A one-processor system would only contain data for CPU 0 (8 bytes, assuming an integer is four bytes long)).**

9.3 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.

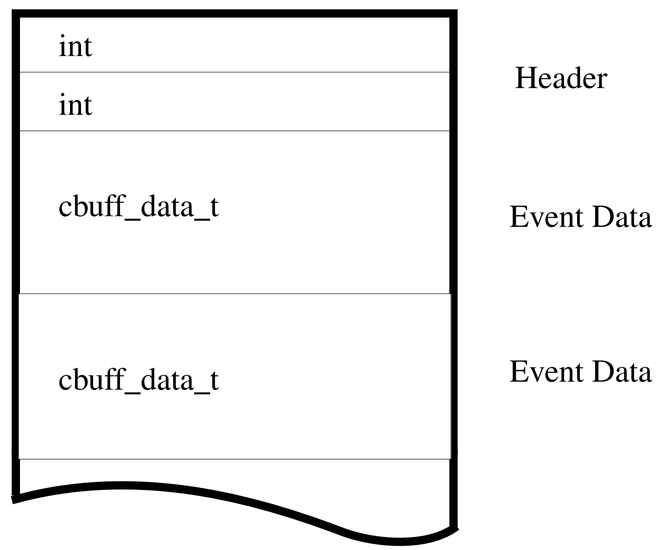


FIGURE 5 **The Cdata file format.**

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

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 (for information on specific bit fields see Section-?).

TABLE 3. 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

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-level probes will be ignored if the p_user_event probe is disabled.

APPENDIX B How To Tips

This Appendix 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.

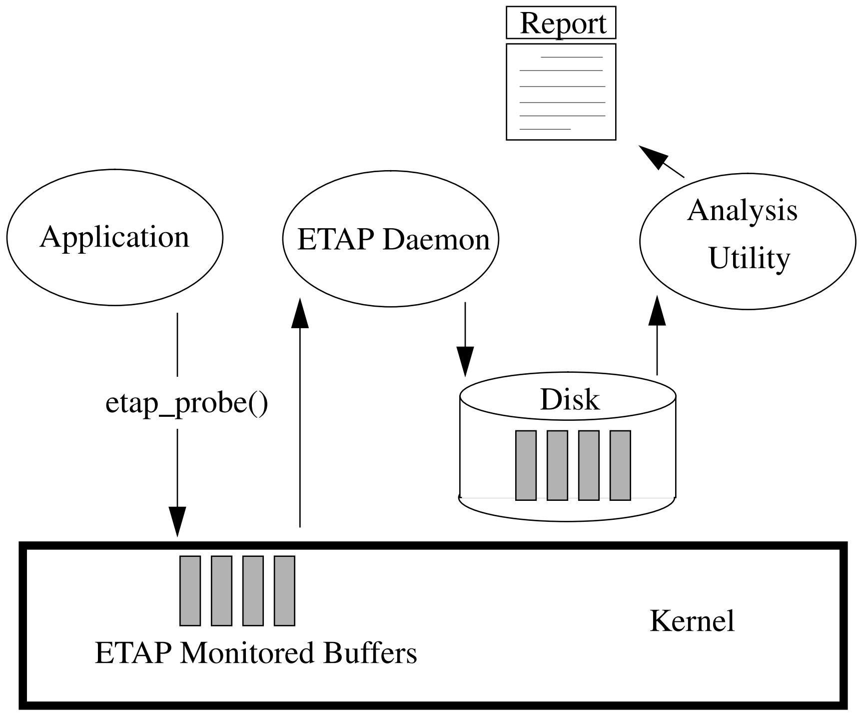


FIGURE 6 **This picture shows the flow of application specific event data from the application to report form.**

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)

APPENDIX C 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 longmax_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);
}
```