State Abstraction-based Synchronization for Thread Libraries

Atsuo OHKI[†] and Yasushi KUNO[††],

Graduate School of Business Sciences, University of Ttsukuba, Tokyo, 3-29-1 Otsuka, Bunkyo-ku, Tokyo 112-0012

Summary
"State Abstraction" is a framework in which states of a data structure (or an object) are divided into small number of exclusive sets ("Abstract States," or AST), and current abstract state is explicitly managed by the code. In "State Abstraction-based Synchronization" (or AST-sync), abstract states of the data structure guarded by a critical region are similarly managed, and an activities can enter the region only when current state is included in the pre-specified set. For example, in a bounded buffer, "get" operation can proceed only in "full" or "mid" state, but not in "empty" state. AST-sync can express conditional synchronization in concise and readable manner, and can be efficiently implemented. Today, thread libraries are widely used for concurrent programming on shared-memory multiprocessors. On POSIX thread and other well-known thread libraries, condition variables are used for conditional inter-thread synchronization. However in those APIs, complex conditional synchronization using multiple condition variables are difficult to understand and error-prone. Hence, programming style with single condition variable is widely used. Standard synchronization scheme on Java programming languages is effectively designed likewise. In this scheme, all threads are woken up on each event and have to check for their continuing conditions repeatedly, leading to large overhead. Thus, we are proposing an alternative conditional synchronization scheme and associated API for thread libraries based on AST-sync. In our scheme, every mutex acquisition are guarded by a set of abstract states on which the processing can continue. In this paper, we report our kernel-level implementation of AST-sync based on multiprocessor-capable POSIX thread library bundled in FreeBSD 6.0-RELEASE, and its performance evaluation using micro benchmarks. As the result, we have observed 2--3 times speedups against popular "repeated tests" type synchronization code for some cases.

Key words:
Conditional Synchronization, Thread Libraries, Condition Variables, Guards.

1. Introduction

In systems with concurrency, such as multiprocessor systems or distributed (networked) systems, some synchronization mechanism is needed to ensure correctness of entire computation.

In case of (possibly distributed) shared-memory systems, critical regions are mainly used. In critical regions, only one activity (process or thread) may enter the region at any moment. When second activity tries to enter the region, its execution is automatically delayed until the first activity leaves the region.

However in some cases, execution of the activity should be delayed even when no other activity is in the region. For example, if the region encloses a bounded buffer, "getter" activity must delay execution when the buffer is empty. The execution of the activity will be delayed until any "putter" enters the region, puts data into the buffer (now the buffer is non-empty), and leave the region. This kind of synchronization is called "conditional synchronization" because execution is delayed until some specific condition (buffer becomes non-empty in the above example) is established.

Condition variables [4], semaphores [5], conditional critical region (CCR) [3] are well known standard mechanisms for conditional synchronization. However, each of them has their own drawbacks. A semaphore is a kind of counters, thus it is applicable only when the "condition" is expressed as a count.

A condition variable represents a "waiting queue" on which activities can be put to sleep and woken up afterwards. Each "condition" (e.g. buffer full, buffer empty and so on) corresponds to one condition variable. Therefore, the code tends to become complex when multiple such condition exists. Moreover, if there are overlaps among conditions, condition variables cannot be used. More accurately, the implementation must permit one activity to sleep on multiple condition variable "at the same time." However we do not know any condition variable implementation with such functionality.

A conditional critical region is a critical region associated with boolean expression ("guard expression"). As the conditions are directly expressed as boolean expressions, this scheme is general and comprehensive. On the other hand, an activity must evaluate a guard at each (trial-) entry to the region (repeated tests), resulting higher overhead.

Herlihy [1] has eliminated the overhead by evaluating guards on transactional memory, but his scheme requires to implement all memory access for guard evaluation as transactional memory access. Such strong constraint limits applicability of his scheme.

Manuscript received March 15, 2007
Manuscript revised June 10, 2007

The authors have proposed an alternative (simple and comprehensive) conditional synchronization mechanism, namely abstract state-based synchronization (AST-sync), and have been investigating its application to object-oriented languages[8][9].

Our scheme can also be incorporated to traditional thread libraries (e.g. POSIX thread) in a portable manner. In [10], we have presented AST-sync thread API implemented on top of POSIX API. Using AST-sync, conditional synchronization among threads can be expressed in a simple and straightforward manner. However, portable implementation uses repeated test (as in CCR) internally, so its performance is similar to CCR.

Therefore, we have developed new, kernel-based implementation of AST-sync. We have modified FreeBSD 6.0-RELEASE operating system kernel and bundled POSIX thread library, so that AST-sync is directly handled by the kernel. We have measured the performance of our new implementation, compared them against traditional schemes, and got the favorable result.

In section 2, we explain concepts of abstract state and abstract state-based synchronization (AST-sync), along with the thread library API we have developed. In section 3, we describe our kernel-based implementation of AST-sync mechanism. In section 4, performance evaluations and their results are shown. Finally in section 5, the summary and conclusion is described.

2. Abstract State-based Synchronization and Thread Libraries

2.1 Abstract States and Synchronization

In object-oriented languages, an object might have several different states. The state of an object is represented as the set of instance variable values. However, as an object corresponds to abstract data type (ADT) value, users of the object are not concerned with detailed state differences.

For example, in a bounded buffer object, data currently stored inside and their ordering are part of its state. However, users of the buffer are almost never concerned with such details; they are mostly concerned with "whether the buffer is empty, partially filled, or completely filled."
Therefore, we propose to explicitly declare set of those "states" as in `{empty, mid, full}` and concentrate only on difference among those "states." These "states" are actually abstractions of the objects' detailed internal states described above. Hence, we call those "states" as "abstract states."

In ordinary object-oriented languages, it is widely adopted conventions that programmers (or designers of classes) prepare predicate methods such as `isEmpty()`, `isFull()`. These predicate methods convey information as to which abstract state the object is currently in. This scheme has the following drawbacks:

1. Not all boolean-valued method corresponds to abstract state differences. Therefore, what are the set of abstract states and how they can be examined is not always clear.
2. It incurs method invocation overhead (method inlining and other optimization might eliminate this overhead).
3. Boolean expression inside the predicate methods needs be evaluated every time, even when abstract state has not changed.

In our proposal, abstract states are represented by small integer values, and the current state value is stored in the fixed place of an object. When the object's abstract state changes, the object is responsible in updating its abstract state value correctly. From outside the object, this fixed place can be examined to determine abstract state of the object in an efficient manner.

So far we have explained the concept of abstract states. Next we explain abstract state-based synchronization (AST-sync). In object-based concurrent systems, each critical region guards corresponding object from concurrent access.

With the above assumptions, conditional synchronization will delay entry to the region until corresponding objects' abstract state becomes the one of specified set. This is because abstract states captures state difference visible (what matters) to the user of the object. If a synchronization condition does not correspond to abstract state differences, then one should reconsider set of abstract state for the object. For example, "putter" is delayed when the buffer's abstract state is `full`, and can proceed when the state becomes `mid` or `empty`.

Based on the above observation, abstract state-based synchronization (AST-sync) is formalized as follows:

1. A critical region is associated to an object with abstract states.
2. Upon entry to a critical region, an activity present set of abstract states ("entry set") for which it can proceed. If the object is not in one of those states, entry is delayed.
3. If the objects' state is (or becomes) contained in the entry set, the activity enters the region. State of the

object is temporally set to null -- which means that object is locked and no other activity can enter the region.

4. When the activity leaves the region, the objects' state is restored (to previous value, or to new value corresponding to the state change). At this time, when some activity is waiting and restored state is in its waiting set, the activity is woken up and enters the region. Note that we make no assumption on region entry orderings when multiple activities are waiting.

When the number of abstract state does not exceed the number of bits in the hardware word, each state can be assigned to single bit within the word, and entry condition can be tested efficiently by "and" mask operation [8]. The scheme can also be extended to the case where class inheritance is being used [9].

2.2 POSIX Thread API

As described above, AST-sync is suitable to be used with thread libraries. In this way, concurrent programs written in C or C++ languages can obtain full benefits of AST-sync.

We have based our AST-sync thread API on popular, widely used POSIX thread [6] API. Firstly, we describe bounded buffer example in POSIX thread API. In POSIX thread, critical regions are associated to mutex lock, and conditional synchronization is implemented using condition variables. In the following code, `mutex` is mutex lock type, and `nonfull`, `nonempty` is condition variable data type.

```
void put(T p) {
   pthread_mutex_lock(&mutex);
   while( [[[ the buffe is full ]]] )
     pthread_cond_wait(&nonfull, &mutex);
   [[[ store data to the buffer]]]
   pthread_cond_signal(&nonempty);
   pthread_mutex_unlock(&mutex);
}
void get(T *p) {
   pthread_mutex_lock(&mutex);
   while( [[[the buffer is empty]]] )
     pthread_cond_wait(&nonempty, &mutex);
   [[[ extract data and put to *p ]]]
   pthread_cond_signal(&nofull);
   pthread_mutex_unlock(&mutex);
}
```

When a thread sleeping on a condition variable is woken up, it seems as if the waiting condition is satisfied and repeated tests of the conditions are unnecessary. However, in POSIX specification, some other thread might obtain lock before the woken thread enters the region, so repeated test is mandatory (although looping case will be quite rare).

Using multiple condition variables is apparently complex and not easy to read. Therefore, many programs use only one condition variable and `pthread_cond_broadcast` for signaling. With broadcast, all threads are woken up, enter the region one by one and test for the condition. In most cases, only one thread is successful in the test and proceeds; others are put to sleep again.

We call such style as "repeated tests." In repeated tests, entry to the critical region is controlled solely by the boolean expression (guard), thus it can be seen as a kind of CCR implementation. Java have only one condition variable per lock (=object), thus repeated tests is widely adopted.

2.3 AST-sync API for Thread Libraries

In this section, we explain AST-sync based thread API we have designed. API itself is shown in table 1. We represent abstract states as suggested in the end of the section 2.1. Therefore, number of abstract states must not exceed number of bits in an `int` value (32 or 64). We think this number as being sufficient, for the number of "abstract" states (which programmers have to investigate one by one in their code) should not be so large.

Table 1: AST-sync based thread API

API	function
`int ast_mutex_init(struct ast_mutex *m, int state)`	Initialize mutex
`int ast_mutex_destroy(struct ast_mutex *m)`	Destroy mutex
`void ast_mutex_enter(struct ast_mutex *m, int mask)`	Enter regin
`void ast_mutex_exit(struct ast_mutex *m, int state)`	Leave region

In original POSIX thread, evaluation of region entry conditions (guards) has to be placed inside the critical region to prevent interference among threads. If repeated tests are used, repeated entry to the region leads to increased overhead.

In AST-sync, an entry condition is a set of abstract states, and is represented by single mask value. Therefore, upon mutex entry we hand this mask value to the library call `ast_mutex_enter`, and synchronization is all handled within the call.

Upon exit form the region, next abstract state have to be established, so we hand the new state value to `ast_mutex_exit`. Additionally, guarded object have to be in some well-defined state upon startup, so we hand the initial state to `ast_mutex_init`.

Using the designed API, bounded buffer example will be written as in the following (initial state is assumed to be S_EMPTY, and buffer size is assumed to be larger than 1).

```
void put(T p) {
  ast_mutex_enter(&mutex, S_MID|S_EMPTY);
  [[[ store data to the buffer ]]]
  if( [[[ the buffe is full ]]] )
    ast_mutex_exit(&mutex, S_FULL);
  else
    ast_mutex_exit(&mutex, S_MID);
}
void get(T *p) {
  ast_mutex_enter(&mutex, S_FULL|S_MID);
  [[[ extract data and put to *p ]]]
  if( [[[the buffer is empty]]] )
    ast_mutex_exit(&mutex, S_EMPTY);
  else
    ast_mutex_exit(&mutex, S_MID);
}
```

Note that after ast_mutex_enter is returned, buffer is already in the desired state and can be handled immediately. On the other hand, new (correct) abstract state has to be established when releasing the lock through ast_mutex_exit.

In summary, code among ast_mutex_enter and ast_mutex_exit forms a critical region, and conditional synchronization is controlled through state values (or masks) handed to API.

3. Implementation

We first implemented AST-sync API on top of POSIX thread API in a portable manner, but its performance is similar to repeated tests. To attain better performance, we have included AST-sync functionality in the FreeBSD kernel. We have based our implementation on FreeBSD 6.0-RELEASE and bundled POSIX-compatible thread library.

FreeBSD had three implementations of POSIX-compatible thread libraries, as in the followings:

1. A pure user-level thread implementation (libc_r).
2. An implementation in which kernel threads and user threads have 1-1 correspondence (libthr).
3. An implementation in which kernel threads and user threads have M-N correspondence (libthread).

This time, we have chosen (2) as the base of our AST-sync implementation. (1) is user-level only implementation and cannot use multiple CPUs, thus was not suited to our research. (3) have to manage complex correspondence between user and kernel level thread, so was not chosen this time.

Below, we explain how we have modified libthr implementation and kernel functionalities used by this implementation to incorporate AST-sync.

Figure 1 shows the data structure in the user data space related to the modification. pthread_mutex is the mutex data structure of the original implementation. umtx_t field in the data structure is accessed from the lock service inside the kernel. Thus, we call umtx_t field as “kernel lock” field.

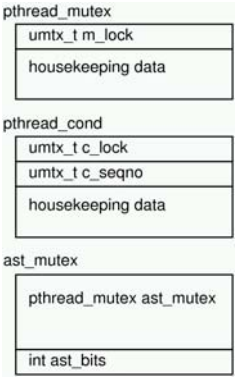

Fig.1 User-space Data Structure

pthread_cond is the condition variable data structure of the original implementation. This structure includes two umtx_t field, one for exclusive access to pthread_cond structure, and the other for exclusive access to queue structure (explained shortly) inside the kernel.

ast_mutex is a new data structure for our AST-sync implementation. The structure consists of original pthread_mutex data structure and extra one word for the current “Abstract State” value.

Figure 2 shows the related data structures in the kernel space. umtx_chain is a hash table; its entries are head of double-linked list data structure. When the numbers of mutex and condition variables are not large, each slot of the hash table will corresponds to single mutex or condition variable (or is unused).

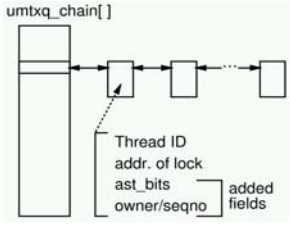

Fig.2 Kernel-space Data Structure

In the original kernel, each element of the double-linked list included thread ID and address of the kernel lock (`umtx_t` data structure) in the user space. To implement AST-sync functionality, we have added two more field as in the following:

- `ast_bits` --- Mask value representing the set of abstract states for which a thread is waiting.
- `owner` --- Field for holding a value used while obtaining `umtx_t` lock, when the waiting thread is being woken up.

These fields are used only when a thread enters wait status using AST-sync functionality. Additionally, one of the field is used to remove bugs (described later) on condition variable synchronization. The field stores the count of signal/broadcast at the conditional wait entry, and is used to verify that no anomaly in wakeup order exists.

When inserting waiting thread to the double-lined list, insertion either at the head or at the tail is possible (no difference in processing cost). Original implementation has chosen head as the insertion point. This is perhaps more efficient, because choosing recently slept thread increases possibility of cache reuse. Thus, we have not changed the choice.

Kernel call from within thread library uses dedicated system call `_umtx_op()`. It had following four commands in the original kernel:

- `UMTX_OP_LOCK` --- Locks `umtx_t`.
- `UMTX_OP_UNLOCK` --- Releases `umtx_t`.
- `UMTX_OP_WAIT` --- Sleeps on `umtx_t`.
- `UMTX_OP_WAKE` --- Wakes up threads sleeping on `umtx_t`. As a parameter, the number of threads to wake up is specified.

To implement AST-sync, we have included two additional commands as in the followings:

- `UMTX_OP_WAIT2` --- Sleeps on `umtx_t` with AST-sync. As a parameter, state mask is specified.
- `UMTX_OP_WAKE2` --- Wakes up thread sleeping on `umtx_t` with AST-sync. As a parameter, state value is specified.

The latter command uses linear search over the double-linked list and wakes up the first thread which matches mask condition. We could eliminate linear search by using more complex data structure, but as a result of evaluation (described in the following section), overhead of current simple implementation is negligible with moderate (about 500) numbers of waiting threads.

In the original implementation, `pthread_mutex_lock` updated mutex data structure (in the user space) and acquired `umtx_t` in one place. However, in AST-sync, `umtx_t` is not released but handed to the thread being woken up. Therefore, we separated function to update mutex data structure and `umtxt_t` operation.

After the modification (during evaluation), we have found that original `pthread_mutex_lock` algorithms was not correct. Actually, the code first released `umtx_t` (to guard mutex data structure) and then immediately slept on `umtx_t`. However, as this "release and sleep" was not atomic, interference could occur (and had occurred in our micro benchmark evaluations).

Therefore, we modified this part of code to enter the kernel while holding the first `umtx_t` and "release and sleep" performed indivisibly in the kernel. As the result of this modification, performance of condition variable synchronization was improved. This is perhaps due to the decrease in system calls (from original two to one per condition variable operation). For the fair comparison, we have used this "improved" version in the following evaluation.

Table 2: # of modified lines

File	# Modified lines	#Total lines
thr_mutex.c	60	1678
thr_cond.c	16	344
thr_umtx.c	22	80
thr_umtx.h	3	87
kern_umtx.c (kernel)	181	772

We have modified four files among 60 files in the thread library (`libthr`), and one file in the FreeBSD kernel. Details are shown on table 2. Workload was full three days by one of the authors, including reading relevant codes in the original library and kernel code. Additionally, the bug explained above required one week to track and fix.

4. Evaluation

4.1 Micro Benchmark

To evaluate the modified library, we have measured performance of following micro bentimarks:

- Standard bounded buffer, which have three states `{empty, mid, full}`. Putter threads are delayed while the state is `full`, and getter threads are delayed while the state is `empty`.

- Bounded buffer with a watermark, having states `{empty, midlow, midhigh, full}` (clients can also see if the buffer is filled halfway or not). In this benchmark, the third kind of thread, namely "marginal putter" is added. Marginal putter threads can proceed only when the status is either `empty` or midlow.

In each of the cases, one buffer with the capacity of ten data is used, while the numbers of each kind of threads are chosen to represent typical conditions for real systems.

Following implementations of the bounded buffer are used:

- Condition variable-based implementation --- Implementation with the framework presented in section 2.2. As explained previously, condition variables cannot be used when there are overlaps among conditions. Thus, wartermarked buffer cannot be implemented in this scheme.
- Repeated tests --- Uses one condition variable; all threads are woken up (broadcast) on every state changes and tests conditional expression repeatedly. (Today's widely used scheme)
- Portable implementation of AST-sync --- AST-sync API is implemented on top of POSIX Thread API. Repeated tests are used internally.
- Kernel implementation of AST-sync --- The core of AST-sync API is implemented inside the kernel. (The one we have explained in the previous section.)

For each of the cases, we have measured performance for both small and large number of waiting threads. The former is represented by the case in which the numbers of getter and putter threads are the same, while the latter is represented by the case in which only one getter thread exists.

In the following, (N_wxC_w:N_rxC_r) means N_w putter threads each putting C_w data combined with N_r getter threads each getting C_r data. For watermaked buffer, (N_wxC_w:NvxC_v:N_rxC_r) means likewise, with N_v and C_v representing the number and put count for marginal putter (refrain from putting when the buffer is more than halfway filled).

Measurement was done in Pentium II 350MHz 2CPU machine. GCC 3.4.4 with -O4 optimization is used for compilation. 100 measurements were performed for each case. In the tables 3-6, average of 100 measurements and standard deviation (in parenthesis) is presented. All time units are in seconds.

Table 3: Bounded buffer with small # of waits (500x100:500x100)

Kind. impl.	User time	Syst. time	Elaps. time
Cond. vars	1.273 (0.100)	12.277 (0.176)	7.16 (0.100)
Repeated tests	3.133 (0.942)	34.181 (11.461)	20.26 (6.805)
Portable AST	3.080 (0.881)	33.637 (10.097)	19.91 (6.012)
Kernel AST	0.601 (0.071)	8.269 (0.264)	6.41 (0.203)

Table4: Bounded buffer with large # of waits (500x100:1x50,000)

Kind. impl.	User time	Syst. time	Elaps. time
Cond. vars	1.081 (0.087)	9.872 (0.244)	5.80 (0.106)
Repeated tests	2.713 (0.254)	28.684 (2.230)	16.99 (1.356)
Portable AST	2.785 (0.215)	29.366 (1.890)	17.40 (1.128)
Kernel AST	0.530 (0.060)	7.340 (0.129)	5.75 (0.083)

Table5: Watermarked buffer with small # of waits (250x100:250x100:500x100)

Kind. impl.	User time	Syst. time	Elaps. time
Repeated tests	4.161 (1.065)	46.892 (13.262)	27.83 (7.865)
Portable AST	3.934 (0.894)	44.130 (11.312)	26.19 (6.698)
Kernel AST	0.594 (0.073)	8.261 (0.229)	6.38 (0.169)

Table6: Watermarked buffer with large # of waits (250x100:250x100:1x50,000)

Kind. impl.	User time	Syst. time	Elaps. time
Repeated tests	3.158 (0.249)	33.392 (2.216)	19.82 (1.328)
Portable AST	2.924 (0.198)	30.858 (1.562)	18.27 (0.930)
Kernel AST	0.539 (0.060)	7.432 (0.126)	5.83 (0.080)

From the tables, we can observe the followings:

- Performance of repeated tests and portable AST is largely equivalent. This is the expected result, as portable AST is implemented using repeated tests inside.
- Comparing condition variables and kernel AST, kernel AST is a bit superior both in user time and system time. This might come from the fact that both user code and kernel code is simpler in kernel AST.
- In spite of the above, superiority of kernel AST decreases when the numbers of waiting threads are large. This might come from the fact that current kernel AST implementation uses linear search over double-linked list on conditional wakeup, meaning search overhead proportional to the number of waiting threads. (Note: data on table 4 is uniformly smaller compared to table 3 because the number of getter threads is decreased from 500 to 1.)

- Compared to kernel AST, repeated tests (corresponds to CCR) and portable AST incurs four to five times overhead. This is the expected result, as kernel AST wakes up only one thread which is guaranteed to proceed, while repeated tests or portable AST wakes up all (500) waiting thread to perform conditional expression evaluation or bit mask test, which will evaluate to false on most cases.
- In watermaked buffer cases, repeated tests and portable AST shows performance degradation. This is probably due to the fact that more states have to be checked and kind of threads are increased. In contrast, kernel AST shows no performance degradation in spite of the above facts.

Also note that, condition variable-based synchronization cannot be used for overlapped conditions (as in wartermarked buffers), but AST-sync does not have such restriction.

4.2 Data-Thread Correspondence

To examine more details, we have assigned numbers 0-99999 to the data transferred (in time order) and numbers 0-499 to getter and putter threads, and plotted points indicating which getter and putter handled which data (for "small number of waits" cases). Figures 3, 4 and 5 shows plots for condition variables, repeated tests and kernel AST respectively. In each, vertical axis corresponds to data number and horizontal axis corresponds to thread number (left half shows putter status and right half shows getter status).

Examining these plots, repeated tests shows the tendency of randomized thread assignment because all threads are woken up and fight for control for each datum. Condition variables show more orderly correspondence and locality, but still show some randomness. On the contrary, kernel AST shows good locality, which indicates decreased inter-thread contentions.

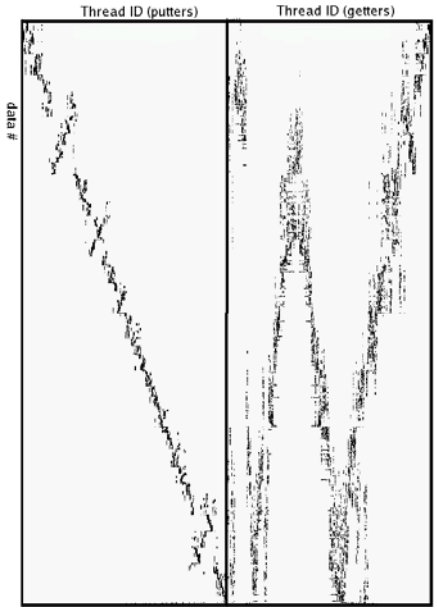

Fig.3 Thread # versus data # plots (Cond. vars)

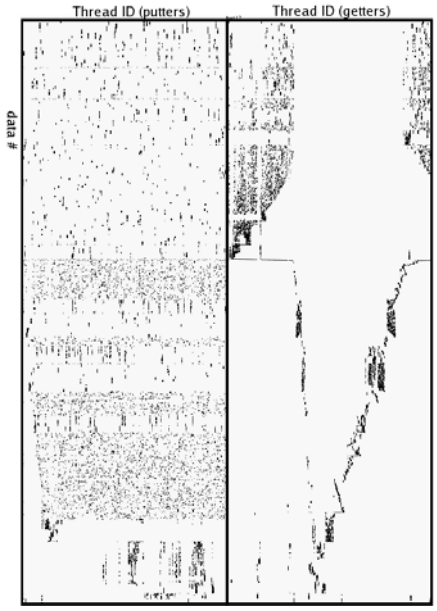

Fig.4 Thread # versus data # plots (Repeated tests)

The reason for such locality can be explained as follows. When a putter (call this thread "A") is suspended due to full buffer, one of the getters (call this thread "B") is selected and resumed, which will eventually empty the buffer and suspends. Then, last putter suspended (thread "A") is resumed. When "A" blocks again, "B" will be resumed again. This pairing will continue until one of

them finishes its job or some global randomization effect (e.g. process swap out or such) occurs.

This kind of coroutine-like behavior leads to good locality and better performance. However, note that the code itself is general one and no thread paring is explicitly coded.

Also note that the above effect is the result of LIFO nature (suspended thread is inserted to the head of ready queue) of thread scheduler used for the experiments.

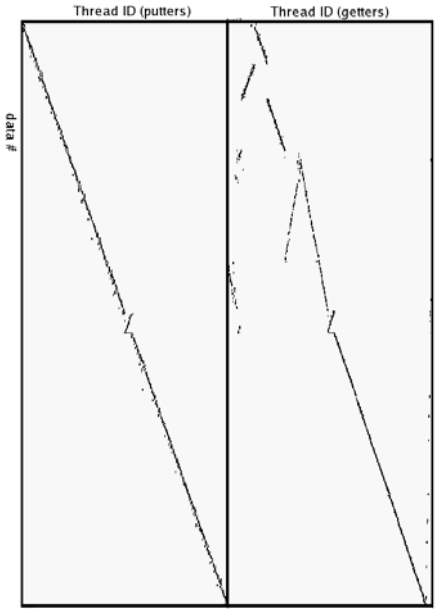

Fig.5 Thread # versus data # plots (Kernel AST)

4.3 Benchmark with More Logical CPUs

To investigate cases for larger number of CPUs, we have also conducted measurements on Pentium D 3.2GHz 2CPU machines with HyperThreading off (the number of logical CPUs is 2) and on (the number of logical CPUs is 4). The results are shown on tables 7 and 8.

When the numbers of logical CPUs are 2, obtained data are mostly similar to the above result. However, data for 4 logical CPUs shows different trends from the above result when the number of waiting threads are small (table 7) and large (table 8).

Small number of waiting threads:

- For condition variables, repeated tests and portable AST, 3-4 times increase in system time is observed. On the contrary, kernel AST show only 20% increase in system time.
- Repeated tests and portable AST show 50% increase in user time, while on condition variables and kernel AST only slight increase is observed.
- For condition variables, repeated tests and portable AST, 100% (twice) increase in elapsed time was observed. On the contrary, kernel AST showed only 10% increase in elapsed time.

Table7: 2 vs 4 CPU: bounded buffer with small # of waits (1,000x1,000:1,000x1,000)

Kind. impl. and # CPUs		User time	Syst. time	Elaps. time
Cond. vars	2	5.189 (0.206)	59.255 (1.250)	33.58 (0.757)
	4	6.252 (0.448)	192.580 (11.929)	65.82 (3.523)
Repeated tests	2	6.658 (1.267)	85.804 (18.400)	49.19 (10.964)
	4	10.267 (1.305)	292.771 (37.942)	105.80 (12.686)
Portable AST	2	6.884 (1.508)	86.683 (23.160)	50.09 (13.813)
	4	10.277 (1.214)	290.425 (34.602)	104.98 (11.606)
Kernel AST	2	2.110 (0.113)	45.436 (1.634)	31.00 (0.874)
	4	2.200 (0.126)	54.224 (1.002)	33.01 (0.519)

Table8: 2 vs 4 CPU: bounded buffer with large # of waits (1,000x1,000:1x1,000,000)

Kind. impl. and # CPUs		User time	Syst. time	Elaps. time
Cond. vars	2	4.763 (0.205)	44.719 (1.754)	26.46 (1.044)
	4	5.581 (0.226)	148.897 (3.933)	53.02 (1.271)
Repeated tests	2	14.599 (1.422)	196.267 (19.348)	116.47 (12.232)
	4	6.892 (0.252)	208.172 (4.350)	76.04 (1.336)
Portable AST	2	14.044 (1.223)	185.283 (15.694)	110.19 (9.840)
	4	7.013 (0.250)	206.813 (5.044)	75.59 (1.538)
Kernel AST	2	2.022 (0.125)	37.319 (0.622)	30.40 (0.412)
	4	1.953 (0.121)	56.588 (0.371)	35.22 (0.211)

Large number of waiting threads:

- For condition variables, the trends are the same as in the cases for small number of waiting threads.
- For repeated tests and portable AST, when compared against the cases for small number of waiting threads, system time increases slightly, user time decreased to half, and elapsed time decreased by 20%-30%.
- For kernel AST, when compared to small number of waiting threads, user time remains the same, system time increases by 50%, elapsed time increases by 20%.

On repeated tests and portable AST, number of waiting threads had the large effect. This might be interpreted as follows. When the number of waiting threads is small, upon thread scheduling, probability of "bad luck" (putter scheduled on a full buffer, or getter schedule on empty buffer) will be 0.5 because of the symmetry of the situation. When the number of waiting thread is large, the sole getter will always have his work and runs at his full speed, thus loss of throughput due to "bad luck" does not occur, leading to better performance. Decreased user time on small number of waiting threads also supports our guess.

For condition variables implementation, it was surprising that we observed large increase of system time uniformly. It might be due to useless contention inside the kernel, but we have not found specific reason for this yet.

In total, kernel AST implementation had uniformly minimal elapsed time in a stable manner (standard deviation of measurement was also very small). When the number of logical CPU is increased, kernel AST implementation observed no user time increase. System time increased moderately because of increased kernel workloads, but this increase did not affect much on the elapsed time.

5. Summary

In this paper, we have added AST-sync, a new and easy-to-use conditional synchronization mechanism, onto POSIX compatible multiple CPU thread library on FreeBSD 6.0-RELEASE. Two versions of AST-sync implementation were developed. One is portable AST implementation, which builds on top of existing POSIX API and need no kernel modification. The other is kernel-based AST implementation and required moderate effort for kernel modification. We have evaluated our implementations against existing scheme (condition variables and popular repeated test-style scheme) using micro benchmarks. As the result, kernel-based AST implementation uniformly had the highest performance, good thread locality and was stable (no large performance degradation) when the number of logical CPUs is increased.

References

[1] Tim Harris, Keir Fraser, Language Support for Lightweight Transactions, OOPSLA'03 Proceedings, pp. 388-402, 2003.
[2] Maurice Herlihy, A Methodology for Implementing Highly Concurrent Data Objects, TOPLAS, vol. 15, no. 6, pp. 745-770, 1993.
[3] C. A R. Hoare, Monitors: Toward a theory of parallel programming, International Seminar on Operating System Techniques, A.P.I.C. Studies in Data Processing, vol. 9, Academic Press, pp. 61-71, 1972.
[4] C. A. R. Hoare, Monitors: an operating system structuring concept, CACM, vol. 17, no. 10, pp. 549-557, 1974.
[5] E. W. Dijkstra, The Structure of THE Multiprogramming System, CACM, vol. 11, no. 5, pp. 341-346, 1968.
[6] ISO/IEC 9945-1:1996 Information Technology -- Portable Operating System Interface (POSIX) -- Part 1, IEEE, 1996.
[7] James Gosling, Bill Joy, Guy Steele, Guy L. Steele, The Java Language Specification, Addison-Wesley, 1996.
[8] Yasushi KUNO, Atsuo OHKI, p6: A State Abstraction-Based Parallel Object-Oriented Language (in Japanese), IPSJ Journal, vol. 38, no. 3, pp. 563-573, 1997.
[9] Yasushi Kuno, Solving Inheritance Anomaly Problems by State Abstraction-Based Synchronization, in Jean-Paul Bahsoun, Takanobu Baba, Jean-Pierre Briot, Akinori Yonzezawa eds., Object-Oriented Parallel and Distributed Programming, pp. 167-186, Hermes, 2000.
[10] Atsuo OHKI, Yasushi KUNO, Incorporating State Abstraction-Based Synchronization into Thread Libraries (in Japanese), IPSJ Transactions on Programming, vol. 17, no. SIG 11 (PRO 30), pp. 28-37, 2006.
[11] Butler W. Lampson, David D. Redell: Experience with processes and monitors in Mesa, CACM, vol. 23, no. 2, pp. 105-117, 1980.

Atsuo Ohki received the B.S. and M.S. degrees in Engineering from University of Tsukuba in 1979 and 1981, respectively. During 1981-1989, he worked as an assistant in faculty of engineering, Shizuoka University. He now works as a lecturer at Graduate School of Business Sciences, University of Tsukuba, Tokyo.

Yasuishi Kuno received the B.S. , M.S. and Doctoral degrees in Computer Science from Tokyo Institute of Technology in 1979, 1981 and 1986, respectively. During 1984-1989, he worked as an assistant in the Department of Information Sciences in Tokyo Institute of Technology. He now works as a professor at Graduate School of Business Sciences, University of Tsukuba, Tokyo.