ISA PROJECT

Porting the ANSA testbench onto Chorus/Mix (3.2)

Abstract:

This document details the initial port of the ANSA testbench onto Chorus/MIX running on Compaq386.

Number:	CS/TR-90-52	
Task:	37.01(a) Draft Deliverable	
Date:	August 1990	
Type:	RC	
Classification:	U	(Unrestricted)
Distribution:	G	(General)
Author:	Prakash Ch Das, Rodger Lea	

Contents

1 Introduction

This document presents the work carried out to produce a version of the ANSA testbench optimised to execute on Chorus/MIX. At the outset we had two major goals, to utilise Chorus functionality to provide an optimised version of the testbench, and to gain an understanding of where it would be feasible to concentrate any further work. In addition we view the testbench port as useful input to our continuing development of the COOL environment.

This document compromises three major section, the first section details the changes required to successfully build a 'vanilla' testbench on the Chorus/MIX operating system.

The second section then discusses the changes made to map certain testbench functionality onto the Chorus Nucleus facilities. We have concentrated our efforts on the ANSA task and communication abstractions.

The third section discusses the performance of our optimised version highlighting the gains we have achieved, justifying and explaining these gains and also discussing why all our results weren't as we expected.

Lastly we conclude with some indications of useful areas for further work and some discusiion on the relationship between the ANSA testbench and the COOL environment.

2 Initial port of the testbench onto MIX

The testbench already runs on various versions of Unix including SunOS and HP-UX. Since Chorus/MIX provides the Unix computing environment, the changes required to port the system are minor. Some changes are processor specific while others are due to minor variations that exist between various versions of Unix implementations.

3 Changes

3.1 Header Files

The following header files are added to reflect the Chorus/MIX computing environment.

include/opsys/chorus.h: This file defines operating system dependent options. It is same as the other header files for various Unix systems except that the names **CHORUS386, chorus386** are defined. These defined names are used to include Chorus/MIX dependent code in a C source file through **#ifdef** preprocessor directive.

include/stub/MoveMacs/80386IEEE.h: This file is derived from **80x86IEEE.h** that is used for Intel's 16 bit 8086 processor. The difference is that the **Boolean** and **Enumeration** data types of IDL which is defined by ANSA are mapped to a 32 bit word rather than to a 16 bit word. This change is required because ANSA defines those two data types in terms the C language types, **unsigned long int** and **enum**; a C compiler on a 32 bit processor maps those data types into 32 bit words.

include/capsule/stack/chorus.h: In this file, the functions for context switching ANSA tasks and some related constants are defined. The functions are **stack_switch()** and **stack_dump()** that are simply redefined, in this case, to C library functions **longjmp()** and **setjmp()** respectively using a preprocessor directive.

3.2 Source Code

The following files are modified.

src/capsule/CORE.unix/{mpsudp.c,mpstcp.c}: Two different versions of **struct sockaddr_in** are used in various versions of Unix. The defined constant, **chorus386**, is used to select the the definition which is appropriate for Chorus/MIX, in functions **MPS_sendMsg()** and **MPS_recvMsg()** of both files.

src/capsule/CORE.unix/sysdep.c: In function, **system_init()**, two extra variables **master_trader** and **trader** are introduced. These two variables are initialized to two strings defined by **MASTER_TRADER** and **TRADER**, and the use of these two constants have been replaced by the respective variable. This extra indirection was added because of the conflict between the **gcc** C compiler and the available C library. According to the proposed ANSI standard for C, **gcc** compiler makes string constants read-only. But the **sscanf()** function of the existing C library wants the format string passed to it to be writable as well.(We could have avoided modifying the source code by making all string constants writable by choosing the correct **gcc** compiler option at compile time.)

src/capsule/timesubs.c: This file is not really a integral part of the capsule. It defines a standard time function, **get_time()**, that ANSA application programs can use to measure elapsed time between two different points of execution. This function has been mapped to **u_chorusTime()** system call in the present case.

src/trader/{offerprg.c, services.c}: The include file name for directory data structure for Chorus/MIX is **dirent.h** not **dir.h**. Similarly, the name of the directory structure is **direct** not **dir**. So, the declarations of variables and the name of the include file were changed but not any executable code.

3.3 Makefiles

The standard header files for library functions of Chorus/MIX are available under the directory **/chincl** not under **/user/include**. In **Makefiles.b** of each directory a new variable **STDI** is defined. It is set to appropriate standard directory name by the install shell script, **InstallANSA** while generating the actual **Makefiles** from the templates, **Makefiles.b**.

3.4 Installation Script

The installation shell script customizes the testbench depending on the operating system environment. So, the script has been modified so that it can handle installation in a Chorus/MIX environment. In the script, this option is called **chorus386** or **CHORUS386**. Also, in the script, I added another variable **STDI** to customize the directory where, by default, the standard C header files are found. In this case, it is set to **/chincl** instead of **/usr/include**.

4 Optimisation of the testbench

This part of the document discusses the changes made to replace the user level ANSA tasks in the testbench by the Chorus kernel provided threads and the use of the Chorus IPC facilities.

4.1 Old ANSA Tasks & Threads

Two concepts are defined to represent concurrency in the testbench. *ANSA threads* represent various points of execution in a testbench. An *ANSA task* represents the resources required, the primary resource being stack, to execute an ANSA thread. ANSA threads provide the notion of logical concurrency while ANSA tasks provide the actual concurrency. An ANSA thread that is in execution is always associated with an ANSA task. ANSA tasks are user level entities implemented through a coroutine package, and scheduling between various ANSA tasks is non-preemptive.

Logically, the testbench software starts with several ANSA threads and one ANSA task. There are a *receiver thread* for receiving messages on the communication channels, a *timer thread* to execute procedures at specified time, a *scheduler thread* to schedule ANSA tasks and an *application program thread* to execute the code of the user program. All these threads always share the initial ANSA task. Application programs may create additional ANSA threads. We may term these as the application program threads. The testbench provides a client/server model of computation and additional threads may be created both in the client or server side of a computation. On the client side, DPL concurrent language construct may be implemented by using several ANSA threads. For example, a client may communicate with two different servers using two different ANSA threads. On the server side, additional threads are created implicitly, if the server, at the time of making available its service, indicates that it can handle multiple requests concurrently. More specifically, the server provides a number saying how many threads may be active simultaneously while exporting its interface to the *trader*. The receiver thread that receives messages on the communication channel creates one additional ANSA thread for each request from clients. A program may also create additional ANSA tasks to provide physical concurrency to the additional ANSA threads that it creates.

4.2 New ANSA Tasks

The structure of the testbench remains almost the same as before. But the implementation exploits the facility provided by the operating system. An ANSA task is implemented as a chorus thread. Therefore, reference to ANSA task means a chorus thread with some private data or context. All old ANSA threads of the old testbench remain except the scheduler thread, since chorus threads are scheduled by the kernel. But there is an important difference. All the initial threads do not share the same ANSA task. All of them, that is, the receiver thread, the timer thread, and the initial thread to execute application program are associated with different ANSA tasks. Besides these ANSA tasks, there is one more ANSA task to handle operating system signals. All signals are handled by the thread associated with this ANSA task only. This has been done to simplify the signal handling mechanism in Chorus/MIX environment.

4.3 Overview of Changes

The original testbench implements ANSA task with a user level co-routine package. It employs non preemptive scheduling to switch between various ANSA tasks. In the design of the original testbench, it has a major effect. Because scheduling is non preemptive, one can maintain shared data structure in a global area without any synchronization mechanism. The testbench took another advantage of non preemptive scheduling. The variables that form the context of an ANSA task are global variables and all ANSA tasks share those locations. Thus, context information are passed to all the procedures through global variables making the procedures non reentrant. This design decision made almost all the procedures non reentrant.

In the new testbench, the ANSA task primitives are mapped to thread primitives of Chorus/MIX. Scheduling of chorus threads is done by the kernel. Therefore, mutual exclusion locks are introduced to protect global data structures shared by all threads. The variables forming the context of a task are bundled into one single data structure and the pointer pointing to this data

structure is made part of the context of a chorus thread using the kernel provided *software register* mechanism. This pointer is stored in a *software register* that is saved and restored by the kernel while doing a context switch between its threads. All procedures can access the software register through a kernel call and thereby access the context information. Some other procedures are made reentrant by passing the parameter explicitly as an argument to the procedure and declaring some variables as local instead of global. These changes were straight forward. But some parts of the code implicitly depended on the scheduling mechanism. For example, consider the following piece of code in English:

```
Wake up the sleeping task A;
Set a flag that will be examined by
    task A after waking up;
Yield control to another task;
```

This piece of code will always work where scheduling is non preemptive, since the task A can not run unless the currently running task relinquish control. This is not correct in a preemptive scheduling environment. In that case, the task A may run before the flag can be set by the current task and consequently, A will not find the flag set which will cause the program to behave erroneously. The problem is that the first two statements should be executed atomically. But in the present example, we can fix the error by interchanging the first two statements since we know that task A is sleeping and it will not be woken up by any other task. The fourth item listed under **schedule.c** in section 5 illustrates this point.

Another design decision that is affected with the introduction of preemptive scheduling and making the context of each task private is the policy of allocating memory contiguously in a dynamic manner. The testbench increases memory for shared data structures in a dynamic manner but requires that the existing memory and the newly allocated one be contiguous. This requirement can be achieved sometimes only by copying the existing data to a new location where enough contiguous memory is available. This means that the testbench should update all pointers that pointed to old memory region. This can be done if all such pointers are stored in known global variables, not in any local variables of other tasks. Also, the entire operation of copying and updating the pointers should be atomic. The solution adopted is to allocate memory statically. The amount of memory statically allocated is not large (less than 4k bytes) and should suffice for almost all programs.(The constants, **CHANNEL_LIMIT, SESSION_LIMIT**, and **THREAD_LIMIT** of **include/config.h** should be increased, if more space is required.) The other solution that we can adopt is to allocate memory dynamically but not necessarily contiguously. The procedures that return the pointer to the data structure given its id can keep track of where various pieces of memory are allocated and return the appropriate address. At present, the position of a data structure within an array is used as an id and therefore, those procedures are extremely simple and efficient.

4.4 Nature of Synchronization

Synchronization is fine grained. Mutual exclusion locks are held for short durations in operations like queuing, de-queuing on shared pools of resources. But the lock that protects the fields of an individual data structure may be held long. This is not a problem since the structure of the software ensures that there is likely to be very little contention. For example, consider the server side of an application. It may be serving multiple clients simultaneously. The server encapsulates information from each client in a data structure called **SessionEntry** separately for each client. The receiver thread, the timer thread and the thread executing the application program decide their action based on the state of this structure. But normally all of them do not need it simultaneously. The receiver thread becomes active when it receives a message from a client requesting a procedure to be executed on the server side. After processing the message and updating the **SessionEntry** structure, it will wake up the application program thread to execute

the request. This thread executes the request and then uses the **SessionEntry** structure to send a reply to the client. Thus, normally there is no contention on the **SessionEntry** structure between the receiver thread and the application program thread. The contention may arise if the receiver thread receives another message from the same client while the reply to earlier one is being sent. Generally, this does not happen since the client will not send another request till the earlier one is satisfied. However, the client may send a retry message, if it does not get the reply within some fixed time period. In that case, the contention will arise if at the same time the **SessionEntry** is is being used for sending the reply. Similarly, the timer thread needs the **SessionEntry** structure when it retransmits the same reply to the client, in case, the acknowledgement from the client is not received within some fixed time period, an event that will not happen frequently.

The possibility of deadlock has been investigated. Threads acquire at most two locks simultaneously but they do so in a order to prevent deadlock. First the lock on the **SessionEntry** structure which is mentioned earlier is acquired and then any other lock. The other lock is generally required while performing operations on common pools of resources.

There is a case when a thread needs to update two data structure atomically (the change in the second data structure depends on the change in the first one) and both the structures use two separate locks. Another thread needs to access these two data structure in the opposite order; the second one is accessed first and after accessing that one it decides that the first one should also be accessed to take some action. The second thread does not need to hold both the locks simultaneously otherwise a possibility of deadlock would have arisen. Now, the first thread can not guarantee atomicity since two different locks are used. The solution adopted, therefore, is that it updates only the first data structure and leaves the state of the program temporarily in an inconsistent state by not updating the second one. The second thread tries to do its operation as before since the second data structure is not updated. But when it accesses the first data structure, it checks for the inconsistency and, if found inconsistent, it aborts the operation and state of the program becomes consistent again. The third item listed under **timer.c** of section 5 illustrates this point.

In two cases, a semaphore is used to synchronize between threads. The first case is when thread sleeps waiting for an event to occur. In the second case, a semaphore is used to synchronize between the receiver thread and the application program threads. On the server side, the receiver thread after queuing a request from a client wakes up a application program thread to execute the request through a V operation on the semaphore.

An application program thread of a client sleeps waiting to receive a reply from a server, after sending out a request. When the thread sleeps, the only lock(mutex) that it may be holding is the lock on the **SessionEntry** structure. If it is holding the lock, it releases it before going to sleep. The thread wakes up with the **SessionEntry** lock acquired, if it was holding the lock before going to sleep. It is the responsibility of the thread waking up the sleeping one to handover the **SessionEntry** lock in acquired state. This arrangement is required to guarantee atomicity while changing a **SessionEntry**. Note that this is different from the case when the sleeping thread acquires the lock after waking up by itself. In this case, some other thread, e.g., the receiver thread or the timer thread may acquire the lock before the thread that has been woken up. We do not want this to happen because those threads will not find the **SessionEntry** in a meaningful state. The thread that does the *wake up* operation has partially updated the **SessionEntry**, and wants to make sure that the next thread that gets control on that structure is the one whom it wakes up. This is ensured using the mechanism we have described above. The fourth item listed under **schedule.c** of section 5 illustrates this point.

4.5 Mapping communication onto Chorus IPC

Our second area of work was concerned with utilising Chorus IPC as the transport layer for the REX protocol. The use of Chorus IPC has already been discussed and outlined in a previous

document [Lea89]. The implementation in this case follows similar lines to that work with the major difference being the use of a single thread per ANSA pin.

5 Specific Changes

This section describes the differences in the code of the new version from the old one. This section assumes that the reader is familiar with the code of the old version (ver 2.6) of the ANSA testbench software.

5.1 Mapping to Chorus threads

thread.c:

- **mutexThreadPool** & **mutexThreadQueue** are introduced to synchronize access to the queues **threadPool** and **threadQueue** respectively.
- **thread_queue()** and **thread_dequeue()** takes one more argument of type **mutex** to guarantee atomicity in operation.
- The memory for **threadTable** is allocated statically at the time of initialization rather than dynamically as it was previously done.[section 4.3, last paragraph].
- **thread_select()** returns the pointer rather than updating the global variable **threadIndex** which has been eliminated.
- **makeThread()** does not try to extend **threadTable** if there is no more **ThreadEntry** is available. [section 4.3, last paragraph].
- **thread_dispatch()** does a V operation on the semaphore **semThreadCount** to indicate presence of a thread in **threadQueue**. This is the only place where a V operation is done on this semaphore.
- **thread_nest()** now checks whether the child is still available in **threadQueue**. This is necessary now since preemption can occur.

task.c:

- This file is almost rewritten.
- **TaskEntry** is completely redefined. It defines the the context of an ANSA task. Now, a ANSA task is nothing but a chorus thread with a **TaskEntry** associated with it.
- **task_init()** associates a **TaskEntry** with the initial chorus thread that the capsule starts with to make it an ANSA task. Now, there is no **taskTable**. **TaskEntry** is created dynamically.
- **task_make()** creates a new ANSA task by creating a new chorus thread and a **TaskEntry**.
- **task_dispatch()** waits for an ANSA thread to be queued to **threadQueue**. It executes an ANSA thread taken from the queue and then waits again in the queue. The ANSA tasks created by the user program starts their execution in this function.
- **task_setup()** creates several dedicated ANSA tasks to execute the receiver thread, the timer thread and the signal handling thread.

- The pointer, **taskPtr**, accesses the **TaskEntry** associated with a chorus thread.
- Two new functions are added. An ANSA task suspends its execution by calling **task_sleep()**. Its execution is resumed by another ANSA task calling **task_wake()** with appropriate argument. **task_sleep()** replaces **schedule()** of the old version.
- The deleted functions are: **table_extended()**, **task_queue()**, **task_dequeue()**, **task_select()**, **task_switch()**, **task_setThread()**, **Capsule_Lock_Acquire()**, **Capsule_Lock_Release()** and **Capsule_Lock_Assert()**. These functions are no longer relevant. The first six functions are used for managing the user level co-routine package to implement ANSA task. The last three functions are used for very coarse synchronization, at the *interpreter level* of the *ANSA capsule.*

session.c:

- The new elements added to the **SessionEntry** are: **myBuffer**, **privatePkt**, **privatePktSize** and **mutex**. The first three, used by the *rex protocol* to send fragmented packet and *rex* header only packet are global variables in the old testbench. These have been made part of the **SessionEntry** to make the *rex* code reentrant. **session_makePrivatePkt()** replaces the **switchPrivateBuffer()** of *rex.c.* For cleaning up, **session_deletePrivatePkt()** is newly defined. **mutex** is added to synchronize access to other elements of **SessionEntry**.
- **session_select()** returns the pointer rather than updating the global variable **sessionIndex** which has been eliminated. **sessionIndex** and **channelIndex** are replaced by **taskPtr->session** and **taskPtr->channel** respectively.
- **session_init()** allocates a fixed amount of memory for **sessionTable**. Now, **makeSession()** does not extend the table if the number of **SessionEntrys** are not sufficient. [section 4.3, last paragraph].
- **AlarmQueue** has been eliminated. Its function is replaced by a general purpose timer package. (The code for this was already in the old testbench but the cleaning up of the code related to **AlarmQueue** was not done.)
- **session_free()**, **session_disconnect()** take a **SessionId** as an argument, instead of receiving the parameter in the global variable **sessionIndex**.
- In **session_selectOutgoing()**, **session_selectIncoming()**, **makeSession()**, **decaySession()**, **session_queue()**, and **session_dequeue()**, codes for synchronization has been added.

channel.c:

- **mutex** is added to **ChannelEntry** to synchronize access to its fields **chain** and **concurrency**.
- **channel_select()** returns the pointer rather than updating the global variable **channelIndex** which has been eliminated. Instead, **taskPtr->channel** is used.
- **channel_init()** allocates a fixed amount of memory for **channelTable**. Now, **channel_make()** does not extend the table if the number of **ChannelEntrys** are not sufficient. [section 4.3, last paragraph].
- In **channel_increment()**, **channel_decrement()**, **channel_make()** codes for synchronization has been added.

sysdep.c:

- **system_setTime()** has been added to update the global variable **timer** that contains current time. Now, **timer** is updated only by the timer thread periodically through the above function. **mutexClock** is the lock associated with the variable **timer**.
- Code for mutual exclusion has been added for memory allocation and deallocation. The lock used is **mutexMalloc**. The changed procedures are **system_allocate()**, **system_deallocate()** (alias **system_free()**), and **system_extend()**.
- The processing that keeps the variable **timer** updated in **system_wait()** is no longer required. Currently, the functionality of **system_wait()** is the same as **wait()** of the old version.

timer.c:

- **mutexTimer** is used to lock timer queues. Code for mutual exclusion has been added in **timer_setTimer()**, **freeTimer()** **timer_clearTimer()**, **triggerTimer()**, and **timer_readNextTimer()** while accessing the queues.
- **timer_processTimers()** does the same job as **processTimers()** of **schedule.c** in the old version. **processTimers()** has been deleted.
- In **timer_sessionPLog()** the lock for the specific **SessionEntry** is acquired. It is released in **timer_sessionELog()**. Once we acquire the lock for the **SessionEntry**, we need to check the state of the **SessionEntry** which may have changed in the meantime and consequently it may no longer be necessary to execute the scheduled action. In the old version, this check was not necessary because if some change in the **SessionEntry** affects the timer queue then the timer queue is also updated. Both these actions need to be done atomically. This is difficult to achieve in the presence of preemptive scheduling. In the current version, if we can not achieve this atomicity because two different locks are used for the **SessionEntry** and the timer queue. Hence, the present solution is adopted.[section 4.4, paragraph 3].
- **timer_init()** has been added to initialize the variable **mutexTimer**.

schedule.c:

- We no longer require **schedule()** since scheduling of ANSA tasks is done by the kernel.
- In **schedule_receive()** which is new, the receiver thread loops forever waiting to receive and process a message. Similarly, in **schedule_timer()**, the timer thread loops forever waiting to execute any procedure appearing in its queue at a specified time.
- **processTimers()** has been moved to **timer.c** with a name **timer_processTimers()**. Functionality is the same but code is a little different. Now, **freeTimer()** is not called from inside **timer_clearTimer()**. This is because of the change that we have mentioned in the third item under **timer.c** of this section. Now, we call **freeTimer()** inside **timer_processTimers()** only after processing a **Timer** variable, not before processing it which was done in the old testbench. **freeTimer()** makes a **Timer** variable available for reuse.
- Code for mutual exclusion has been added in **receiveMessages()**, **invoke()** and **invoked()**. Notice that we do not always release the lock for the **sessionEntry** in **receiveMessages()**. We hand it over to the thread that is being woken up to ensure that all the changes in the **sessionEntry** is doen atomically. Similar action is taken in **timer_sessionElog()** as well.[section 4.4, last paragraph].

- In **invoke()**, we can no longer set **replySession** of a **ThreadEntry** after it has been queued for an ANSA task to execute it. This is because, with the introduction of preemptive scheduling, some ANSA task may act on the **ThreadEntry** before **replySession** can be set and the program will behave erroneously. Now, **replySession** is set at the time of creation of a **ThreadEntry** in **thread_dispatch()** of **thread.c**.[section 4.3, paragraph 2].

protocol.c:

- **mutexSequenceNumber** has been added for locking the variable **sequenceNumber**. This locking is done only at one place in **rex.c** while preparing to send a message.
- The startup function of a protocol is now called with its protocol number. In the old version the protocol number was passed in the global variable **protocolIndex** which has been eliminated. Similarly, the startup function of message passing module and **protocol_open()** is called with its protocol number as the argument. The current protocol number has been made part of the context of an ANSA task. Originally, the procedures that used the global variable **protocolIndex** now uses the variable, **taskPtr->protocol**.
- The global variable, **pinIndex** has not yet been eliminated. Only a receiver thread uses this variable, and currently there is only one receiver thread. Hence, it does not create problem. If we introduce more than one receiver thread, we will need to make **receiveMessages()** of **schedule.c** reentrant.

rex.c:

- The global variables **myBuffer**, **privatePkt**, and **privatePktSize** have been made part of the **SessionEntry**. **switchPrivateBuffer()** has been replaced by **session_makePrivatePkt()** of **session.c**. Functionality of the new function is the same as before.
- The procedures of **rex.c** received parameters in global variables like **sessionIndex**, **channelIndex**, and **protocolIndex**. These have been made part of the context of an ANSA task and are accessed as **taskPtr->session**, **taskPtr->channel**, and **taskPtr->protocol**. The corressponding pointers such as **sessionPtr**, **channelPtr** were also global variables. These have been eliminated. All procedures now calculate the required pointer locally, after taking the index from the ANSA task context.

instruct.c:

- All the procedures have been modified in a minor manner. Some statements such as **thread_setSession()** are no longer required. **schedule()** has been replaced by **task_sleep()**. Appropriate code for mutual exclusion has been added. Global variables such as **threadIndex** have been replaced by **taskPtr->thread**.

nucleus.c:

- Only **main()** is affected. We call few more initialization procedures because of the need to initialise the mutual exclusion locks. Those procedures are **binder_init()**, **ecs_init()**, and **timer_init()**. The procedure **system_postinit()** sets the ignore signal handling option for all signals received by the initial chorus thread. This change is required as we have a dedicated chorus thread to handle signals received by a *capsule*. Also, the code related to scheduling has changed. In **nucleus_task()**, **task_make()** is called with one additional argument which is the entry procedure for a newly created ANSA task.

mpsudp.c:

- **MPS_startup()** takes an extra argument, its own protocol number.
- **rem_addr** which was a global variable is made local to both **MPS_sendMsg()** and **MPS_receiveMsg()**.

buffer.c:

- Code for mutual exclusion has been introduced. The lock used is **mutexBuffer**.

ecs.c:

- Code for mutual exclusion has been introduced. The lock used is **mutexEcs**.

binder.c:

- Code for mutual exclusion has been introduced. The lock used is **mutexBinder**.

idcache.c:

- Code for mutual exclusion has been introduced. The lock used is **mutexIdCache**.

5.2 Changes to utilise Chorus IPC

ANSA.h:

- PROTOCOL_LIMIT is made 6 and REX_CHORUS is defined.

src/trader/server.c

- ADDRESS_LIST is initialized properly so that the message passing module mpschorus.c can recognize that it is part of trader module.

src/capsule/sysindep.c

- system_HtoN(), system_NtoH() and crack_trader_string() now handle the type of CapsuleAdr when REX_CHORUS is used.

src/capsule/mix/mpschorus.c

- This MPS module has been rewritten.

src/capsule/mix/schedule.c

- schedule_receive() gets an argument of type PinId.
- receiveMessages() has been replaced by receiveMessage() that gets PinId as an argument. The difference is that now one receiver will not wait for messages to arrive on any pin or socket. There will be a separate receiver thread for each pin.

src/capsule/mix/task.c

- task_setup() now creates one receiver thread per message passing service(i.e. per pin)

src/capsule/mix/protocol.c

- In protocol_open(), the value of pin argument is checked against the constant SET_SIZE. This is done since the value of a pin can be Chorus local identifier of a port not just a file descriptor of Unix which is normally small. The typical value of Chorus local identifier of a port is 18 or 22.

6 Performance Comparison

The new version is found to be faster than the old one. The difference becomes more pronounced when the application programs do i/o or are part of a distributed application using underlying intercapsule communication.

This speed-up is a combination of three factors, firstly the use of multiple threads within the testbench package itself has increased CPU utilisation because of the scheduling algorithms within the Chorus Nucleus. Secondly, the use of system level threads has enabled useful work to be done when one or more threads are blocked on i/o, including communications. Lastly, the Chorus IPC facilities are faster than the equivalent UDP facilities [1]

While comparing the two versions, we have kept all the configuration and customization parameters the same to ensure meaningful comparison.

Before describing the tests, we give a general outline as to how an ANSA application works.

The ANSA testbench supports client/server model of computation. In a distributed ANSA application program, normally there will be two parts: server and client. Each part runs as a separate process. A server exports operations (i.e. loosely speaking, a set of procedures) that clients can invoke. In ANSA, a server exports operations through an intermediary object, known as the *trader* which also runs as a separate process. A client finds an appropriate server for its operations by contacting the trader and after that the client directly communicates with the server. On the server side, there may be several ANSA tasks to execute exported operations on demand from client. We may call them as application tasks. The number of such tasks is controlled by the user program. By default, there is always one application task. In the absence of any invocation from clients, these tasks remain idle. The invocations from clients are put in a central queue. An application task de-queues an invocation from this central queue and starts the operation the client has requested. The application task after completing the operation sends the reply to the client, and then, looks for another invocation in the queue to repeat the cycle. If the queue is empty it becomes idle. There may be more than one application tasks running concurrently. This will be the case, if user program creates additional application tasks and the queue is non empty to keep the application tasks busy.

[1] Currently local IPC is of an order of 3 times faster and remote 1.5 times faster. However, Chorus IPC is currently under redesign and it is expected that future versions will be greatly optimised.

6.1 Test 1:

In this test, the client and the server simply exchange data. This is in fact an implementation of the standard 'echo' test provided in the testbench suite.

All that the client asks from the server is some number of bytes as a reply.The server does not examine the data sent by the client except for finding out how many bytes needs to be sent as a reply.Client and server run on the same m/c. The m/c is a Intel's 80386 based COMPAQ running Chorus/MIX. The maximum size of an individual packet is roughly 1k bytes.

The three versions of the testbench are the original vanilla testbench running on Mix, version 1 which is the version optimised to use Chorus threads, and version 2 which is additionally optimised to use Chorus IPC.

Number of times exchange repeated = 100

Size of data in bytes Sent	Received	Vanilla Version (seconds)	Version 1 (seconds)	Version 2 (seconds)
16	0	2.75	2.75	1.35
512	0	3.10	2.85	1.35
1024	0	3.50	2.90	1.45
2048	0	5.15	4.55	2.1
4096	0	14.70	10.00	3.2
8192	0	29.70	15.70	10
16384	0	52.20	30.95	15
0	16	2.65	2.70	1.3
0	512	3.05	2.85	1.35
0	1024	3.45	3.00	1.4
0	2048	5.30	4.55	2.15
0	4096	15.00	10.00	3.3
0	8192	28.10	15.35	5.75
0	16384	51.15	30.75	11.65
16	16	3.15	2.65	1.35
512	512	3.45	2.95	1.45
1024	1024	4.45	3.25	1.55
2048	2048	6.60	6.35	2.95
4096	4096	20.50	19.60	5.25
8192	8192	49.55	30.65	13.2
16384	16384	106.70	58.90	30.1

6.1.1 Result analysis

This test is simply a test of the communication cost between two ANSA capsules. There are a number of observations. Initially the Vanilla and version 1 results start out the same. This it to be expected as the communication costs far outweigh the actual processing costs. Once the cost of processing begins to increase as a part of the total cost we begin to see the initial speedup brought about by the use of multiple Chorus threads within the testbench library itself. For example 1024 bytes is approximately 17% faster. When the size of the packet exceeds the maximum packet size the gain through using Chorus threads becomes even more apparent, this is because the cost of fragmentation lies in the multiple calls through the communication layers, hence the initial gains of the Chorus version are multiplied. Thus at a packet size of 8192 (47% faster).

When an attempt is made to measure the gain when using Chorus IPC in addition to the Chorus threads we see that small data sizes are approximately 50% faster (16 bytes) and larger packets are approximately 70% faster (16384 bytes).

It should be noted that the comparison above is biased *against* Chorus for large data packets as we restricted our packet size to less than the 64k allowed in order to make a valid comparison. Using a larger packet size will obviously greatly increase the speedup of simple test programs as above.

This information is represented graphically in appendix A.

6.2 Test 2:

The second test was constructed to try and get a more meaningfull comparison by using a more realistic test program. Our intention here was to examine the speed up that we gained when an application was doing some measure of local computation and a some measure of blocking i/o.

In this test, the server is a toy fileserver. Clients can do usual file operations (open, read, write, seek, and close) on the m/c on which the server is running. The client program of this test copies a file from the server m/c to the local m/c using the operations that the server provides.

- Server m/c: COMPAQ-386 running Chorus/MIX
- Client's m/c: SUN workstations running SunOS
- All clients run on different machines [2].
- All copy different files(but of equal length) while testing the effect on the server with more than one client active simultaneously.
- The timings do not include creation time for ANSA tasks. In this test ANSA tasks are not being created dynamically.
- While comparing the old and the new version, clients are the same in both cases. Clients use the old version of the software. Only the server is different.
- NTASKS = Number of application tasks in the server to handle clients' requests.
- NCLIENTS = Number of clients running simultaneously.
- QLENGTH = Average queue length (i.e. number of invocations that are pending in the server) at the time a new invocation is queued.(Meaningful only in the new version. If NCLIENTS is 1, QLENGTH has to be 0. The maximum length of the queue at the time of queueing in a new request can only be 1 less than NCLIENTS. The minimum length can be 0.)
- NACTIVE = Average number of application tasks that are already running (i.e. the processing on a client's request has started but not yet finished) at the time when an application task de-queues an invocation from the central queue.(Meaningful only in the new version. If NTASKS is 1, NACTIVE has to be 0. The maximum number of tasks that can be running concurrently when an idle application task becomes busy is obviously 1 less than NTASKS. The minimum value can be 0.)
- COPYTIME = Time taken by the client to finish copying the file.

[2] except the local figure.

NTASKS	NCLTS	QLEN	NACT	COPYTIME (seconds) vanilla ver		ver 1		ver 2	ver 2 loc
1	1	0	0	7.49		6.41		3.85	1.75
1	4	0.9	0	25.40	28.16	14.37	14.32	-	-
				26.75	27.83	14.38	14.32	-	-
2	4	0.5	0.6	27.34	27.03	14.80	15.10	-	-
				25.92	26.92	15.31	15.42	-	-

6.2.1 Result Analysis

The above table shows that with 1 client. and 1 task clients take less time to copy a file when the server program uses the new testbench. The reduction in time in this case is about 43%. Using Chorus IPC increases this gain to 49% in the remote case and 74% in the local case.

When we use 4 clients and 1 task then we experience a speed up based solely on the use of Chorus threads within the testbench library of between 40% and 50% for each of the 4 clients. This is in line with our previous general results and is to be expected. Again if this test was performed with Chorus IPC within a Chorus domain the speed-up is in the order of 50% to 60%, however we have factored that out of the tests and concentrated on using UDP communications between heterogeneous machines.

The second part of the test attempts to evaluate the speed-up gained by not only using Chorus threads to provide a multi-threaded version of the testbench library, but also to see what gain could be made by using several threads to deal with multiple incoming requests.

The data in the table shows that the running time of the clients has not decreased significantly discounting communications costs In fact even when we performed the test with 8 clients the speed-up was not impressive, in fact in one case we experienced a decrease in speed.

To explain this lack of expected speed-up we need to look closely what is happening in the server program. The two parameters, QLENGTH and NACTIVE provide clues for this unexpected behaviour. The client program is a file copying program. So, almost all of the invocations from clients are for getting a fixed number of bytes(in this case 4096 bytes) through file read operations. If the read operation does not block in the server, it is an operation of short duration. One application tasks is sufficient to handle invocations from several clients. When four clients run on 4 different machines simultaneously, we see that QLENGTH is just 0.9 instead of being somewhere near its maximum possible value, 3. This shows that there is not much scope for more than one application ANSA tasks. When we increased the number of application tasks by 1, the QLENGTH dropped to 0.5. We expect this to drop to 0. This is not happening. This means that application tasks does not become busy immediately even if there is an invocation pending.

Similarly, NACTIVE gives a rough indication of the extent of concurrency among application tasks. The indication is rough because it does not tell us the duration during which these application tasks run concurrently. A high value of this parameter does not necessarily mean clients will get faster response than the case when the server has fewer application tasks. If the duration for which these application tasks run concurrently is short, we will not see any improvement. Ideally, we would like NACTIVE to be maximum(1 less than NCLIENTS). In the present case, it is 0.6 instead of being somewhere near the ideal value, 1. Here, we are probably seeing the effect described above. The file read operation successfully completes without being blocked is an operation of short duration. Hence, this much concurrency does not seem to be sufficient to bring down the running time on the client side.

Thus we are gaining little speed-up because few requests are blocking but are paying the extra cost of Chorus thread creation and thread context switching.

6.3 test3:

In an attempt to gain a better understanding of the performance gains when using multiple tasks to deal with multiple requests we modified our fileserver program to cause it to simulate extra I/O and to block for greater periods of time. This was achieved with a combination of simulated processing, blocking and using the Chorus distributed filesystem to access remote files stored on machines around the network.

The test was carried out with a server program executing on a Compaq386 machine and 4 clients executing on Sun3 machines each using a vanilla version of the testbench library.

Since our only concern in this test is the performance gain through using multiple Chorus threads we have made no attempt to compare the speed-up in comparison to a vanilla testbench capsule. The percentage speed-up would be no different from those already demonstrated. However, the figures in the table below represent the speed-up when an increasing number of threads are created within our fileServer to deal with multiple remote requests.

		COPYTIME (seconds)	
NTASKS	NCLTS	Version 1	% increase
1	4	85.70	-
2	4	43.78	49
3	4	29.82	65
4	4	25.54	70

6.3.1 Result analysis

The table shows that as we increase the number of ANSA tasks that are waiting to deal with incoming requests then the average response time for the 4 clients is reduced. The % increase column gives an indication of the sped-up as a result of an extra task being available in the server. In an ideal situation the speed up would be X2 for two tasks, X3 for 3 and X4 for 4 tasks. In fact in the test figures above we achieve close to the ideal for the first case but as we increase the number of tasks our 'rate of return' falls off. This, as discussed in the previous test is a result both of our overheads for Chorus thread switching and of the number of threads queued at any one point. It would be possible to gain an almost perfect speed up [3] by providing a server that spent a considerable amount of time carrying out I/O operations, however, the simulation we used for this test carries out both simulated I/O and processing, thus we never achieve the ideal situation of our number of active tasks being 1 less than the number of tasks in the system.

This information is represented graphically in appendix A.

7 Miscellaneous

Now, the application program written in DPL is just one chorus thread in a multithreaded program. Therefore, if the application program uses C library functions the program may not work as expected. For example, now memory allocation, if done using **malloc()**, needs to be done using the lock, **mutexMalloc**. A C library function may implicitly allocate memory by calling **malloc()** and thus bypassing the associated lock. In some case, the C function may not be reentrant. For these reasons, ANSA application programs and the testbench should use a special C library which is designed to work in a multithreaded program. Such a library is currently under development at Chorus Systemes.

[3] taking into consideration thread overhead

8 Conclusion

We feel that the above results achieve our first goal of producing a more performant version of the testbench library. However, this speed-up has been gained at the loss of generality because of our use of features of the Chorus operating system.

However, the value of the work lies more in its potential as Chorus evolves. As mentioned above, the Chorus IPC mechanism is beging moved of the OSI stack and onto bare transport protocols, the speed up of this will be significant. Perhaps a more important fact is that the Chorus testbench version now contains sufficiently fine granularity locking to allow true parallelism. The benefit of this will only be fully apparent when the testbench runs on Chorus on multi-processor systems. A port to such a system will require little further work.

The current testbench runs on Chorus/MIX, the MIX environment provides both a full Unix development environment, and more importantly the device handlers absent from the native Nucleus. Whilst this is probably the most practical environment it incurs the extra overhead of using the MIX system interface. For example, although we are using Chorus threads, we create and control these threads via the MIX thread interface which causes an extra level of management. It would be interesting to implementat ANSA as a separate sub-system and evaluate this cost.

Our current version makes full use of the REX protocol suite, it would be possible to dispense with REX and map directly onto Chorus IPC and more importantly Chorus RPC. However, doing so creates two problems, the first which is only an implementation consideration is the degree of inter-dependence between the communications software and the rest of the ANSA library. A second problem is related to intercommunication between heterogeneous systems, by maintaining the REX communication package we are eaisly able to interwork between ANSA capsules on differing hard and software platforms. However, we pay the cost that we map a RPC protocol onto our own IPC layer, which in itself already contains some of the functionality of the lower levels of REX. It should perhaps be considered in the future, how the current testbench communication model can be adapted to offer one or more other interface points that will allow the new generation of operating systems, ie those developed for distributed systems, to utilise their own RPC mechanims.

We are evaluating how we can integrate the results of this work into our main activity, the development of our own ANSA conformant platform (COOL). Currently we are developing a simple bridge from our existing COOL platform to the ANSA testbench which will allow objects to communicate. A more ambitious plan would be to port aspects of the ANSA testbench's upper layers to the COOL platform to allow COOL objects to be viewed as ANSA objects. This would be an interesting area for future development.

Lastly, we have attempted in the test suite to exercise relevent aspects of our version of the testbench, in particular we have tried to isolate communications speed-up, internal testbench speed-up and multi-threaded user applications. However, in all cases the test programs are contrived. It would be an interesting exercise for evaluation purposes to benchmark the testbench with a live piece of software, it is hoped that a member of the consortium concentrating on the application level will do this.

References

[Lea89] Lea, R. "Porting the ANSA testbench onto the Chorus Simulator (3.1)" Chorus systemes Technical report CS/TR-89-56, 1989.