SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 22(3). 265–285 (March 1992)

Implementing Concurrent C

N. H. GEHAN1 AND W. D. ROOME
AT& T Bell Laboratories, 600 Mountain Ave, Murray Hill, New Jersey 07974, U.S.A

SUMMARY

Concurrent C (C++) is a parallel superset of C (C++). Versions of Concurrent C have now been implemented for a variety of uniprocessors and multiprocessors. We first implemented a uniprocessor version of Concurrent C correctly anticipating that it would be relatively easy to extend the uniprocessor implementation to run on multiprocessors. Concurrent C is translated to C. The generated C code contains calls to the C library implementing the Concurrent C run-time system.

This paper describes the 'hard core' details of the Concurrent C implementation: the specifics of process states, the data structures used, the library functions, and the C code generated for various Concurrent C constructs. We also give an overview of the Concurrent C facilities and of the uniprocessor, distributed and shared-memory multiprocessor implementations.

KEY WORDS Concurrent programming Concurrent C++ C++ Parallel programming

INTRODUCTION

Concurrent C (C++) [1,2] is a superset of C (C++) [3,4] that provides parallel programming facilities. A Concurrent C program consists of components that can be executed in parallel. In designing Concurrent C, our primary goal was to provide a tool for distributed programming. The same compiler supports both Concurrent C and Concurrent C++. A compile-time flag determines which language is accepted by the compiler. Unless it is necessary to be specific, we will use the name 'Concurrent C' to refer to both Concurrent C and Concurrent C++.

We first implemented Concurrent C on a uniprocessor for several reasons:

1. It would be easier to implement Concurrent C on a uniprocessor than on a multiprocessor.
2. This would give us quick feedback on the design of Concurrent C and techniques for implementing it.
3. The uniprocessor implementation would be much more portable than a multiprocessor implementation.
4. Finally, we correctly anticipated that the uniprocessor implementation would serve as the basis of the multiprocessor implementations.

The Concurrent C compiler translates Concurrent C (and Concurrent C++) to C, which is then compiled by the local C compiler, and linked with a Concurrent C run-time library and the system libraries.

In this paper we describe 'hard core' implementation details: the specifics of

0038–0644/92/030265–21$10.50

Received 13 May 1991
Revised 21 October 1991

process states, the data structures used, the library functions, and the C code generated for various Concurrent C constructs. We also give an overview of the Concurrent C facilities and of the uniprocessor, distributed and shared-memory multiprocessor implementations.

BRIEF SUMMARY OF CONCURRENT C

A Concurrent C program consists of one or more processes co-operating to accomplish a common objective. Concurrent C provides facilities for declaring and creating processes, process synchronization and interaction, process termination and abortion, priority specification, waiting for multiple events, and so on. Two processes interact by first synchronizing, then exchanging information and, finally, continuing their individual activities. This synchronization or meeting to exchange information is called a *rendezvous.*

Transfer of information during the rendezvous is unidirectional—from the message sender to the receiver. Concurrent C uses the *extended rendezvous* or transaction concept to allow bidirectional information transfer during the rendezvous. After a transaction has been established, the process requesting service is automatically forced to wait until the server completes the requested transaction; the transaction results are then sent back to the waiting client.

Specifically, Concurrent C provides facilities for

(a) defining processes (a process definition consists of a process specification and a process body)
(b) creating processes
(c) specifying the processor on which a process is to run
(d) specifying, querying and changing process priorities
(e) specifying synchronous transactions (for synchronous bidirectional information transfer)
(f) specifying asynchronous transactions (for asynchronous unidirectional information transfer)
(g) ordinary and timed transaction calls (the latter can only be used for synchronous transactions)
(h) delays and timeouts
(i) interrupt handling
(j) waiting for a set of events (using a guarded select statement)
(k) accepting transactions selectively, in a user-specified order
(1) process abortion
(m) automatic collective termination.

For further details about Concurrent C, see Reference 2.

IMPLEMENTATION OVERVIEW

We will first give a high-level description of the UNIX® uniprocessor implementation of Concurrent C and then discuss how we extended it to work on the multiprocessors.

® UNIX is a registered trademark of USL.

Uniprocessor implementation

The UNIX uniprocessor implementation runs on top of the UNIX system and does not require any changes to the UNIX kernel. Each Concurrent C program, which consists of one or more Concurrent C processes, is implemented as one UNIX process. Within the UNIX process, each Concurrent C process has its own stack and machine registers. The Concurrent C run-time library provides a process scheduler which switches between Concurrent C processes by saving and restoring registers. This is done purely at the user level, without invoking the operating system. This internal Concurrent C process scheduler is invoked at process interaction points and at regular intervals (time slicing). The latter ensures that no CPU- or I/O-greedy process starves other processes of equal or higher priorities.

We chose to put all Concurrent C processes in one UNIX process, in preference to putting each Concurrent C process in a separate UNIX process, for several reasons. First, the cost of switching between Concurrent C processes is substantially less than that between UNIX processes. Each UNIX process has its own virtual address space; context switching is slow because changing address spaces is slow (about 1 ms on a VAX 11/780). Switching between Concurrent C processes is fast (about 50 μs) because these processes share the same address space (i.e. the address space of the containing UNIX process); a context switch essentially involves saving and restoring registers. A fast process context switching mechanism encourages programmers to write small, simple, modular processes—just as a fast function call mechanism encourages programmers to write small, simple, modular functions.

Secondly, because all Concurrent C processes are within one UNIX process, they have the same address space. Our implementation exploits this fact; e.g., instead of copying messages between processes, we pass pointers.

Finally, having a separate, user-level scheduler for Concurrent C processes allows us to experiment with different scheduling strategies.

For portability reasons, we do not encourage the use of shared memory between Concurrent C processes. However, shared memory allows efficient (and possibly convenient) data access. Therefore Concurrent C allows processes to use shared memory, but does not provide facilities for synchronizing access to shared memory. It is the programmer's responsibility to ensure consistency of the shared data.

We built our own lightweight process package for Concurrent C, instead of using an existing package, because there were no such packages when we initially implemented Concurrent C. Since then, several packages have become available, but they tend to be machine specific. As yet, no lightweight process package has emerged as a standard that is commonly available on a number of different machines. When (and if) such a package emerges, we expect to change the Concurrent C run-time library to use it.

Local area network (LAN) implementation

Our first multiprocessor implementation was targeted for a set of object-code compatible computers (VAX 11/780s and 8600s) connected by an Ethernet local area network (LAN) and the Berkeley 4.2 BSD UNIX system. Each Concurrent C program is implemented as one or more UNIX processes which are connected by socket streams (full-duplex communication channels), as in Figure 1.

As in the uniprocessor implementation, each UNIX process contains a set of Concurrent C processes, and an internal scheduler switches between those processes. Normally each UNIX process runs on a separate physical processor, and can be regarded as a virtual processor. Of course, for testing in a single processor environment, we might put several UNIX processes on one processor. Each UNIX process executes a copy of the same load module. The main process is executed only on the 'master' UNIX process—the one on which execution of the Concurrent C program was initiated. When a new Concurrent C process is created, it is assigned by the user to a UNIX process, and remains on that UNIX process until it terminates (i.e., we do not support process migration).

Each UNIX process contains a Concurrent C process which handles messages from other processors—a 'message reading daemon' process. This daemon process periodically examines all input streams to the UNIX process. If any messages are waiting, it takes the appropriate action. In most cases, these actions involve putting messages and results on appropriate queues and changing the states of the Concurrent C processes. Like the uniprocessor implementation, the distributed implementation does not require any changes to the UNIX kernel.

Our initial prototype implementation uses a star configuration. Eventually, we plan to provide full interconnection between the UNIX processes. This will allow processes on different processors to directly send messages to the appropriate processor without having to go via a master processor. Also, this will prevent the master processor from becoming a bottleneck.

Shared memory (SM) implementation

The shared memory multiprocessor (Figure 2) consists of several single board computers (SBCs) and some memory, all on a common global bus. In addition, a host UNIX system can access the global bus; it acts as a debugger, monitor, and I/O server.

Each SBC consists of a microprocessor and some local (on-board) memory. Accesses to the local memory use the SBC's local bus, and do not tie up the global bus. If the microprocessor accesses a memory address that is not within the local memory section, the SBC places that reference on the global bus. We have used several different commercially-available SBCs, mostly with Motorola 68020 or 68030 processors, and with on-board memory sizes from 1 to 8 megabytes. Each SBC's local memory is dual-ported, in that the local memory is assigned an address range on the global bus, and the SBC responds as a memory slave to any bus request for those addresses. Thus the SBCs can reference each others local memory. Devices,

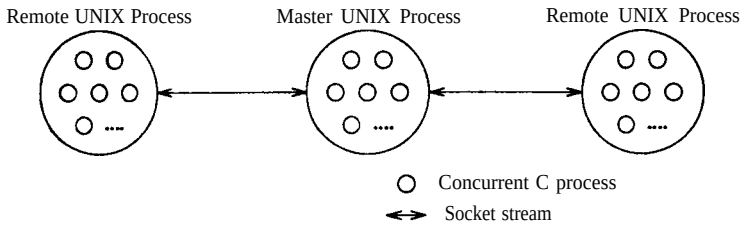

Figure 1. LAN multiprocessor

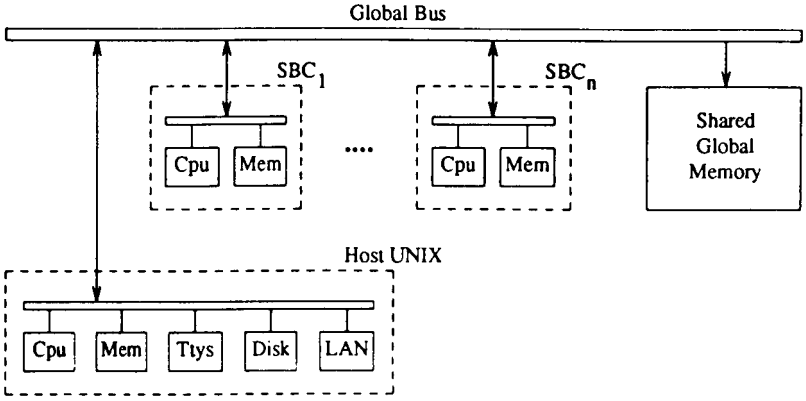

Figure 2. Shared memory multiprocessor

such as disk controllers, can also be connected to the bus. These devices can be controlled by Concurrent C processes: they can access the device control registers, handle interrupts, etc.

The host UNIX system acts as a monitor and debugger. Because the SBCs' local memories are dual-ported, programs running on the host can examine or alter them. The host can examine and alter the global memory. Programs on the host provide UNIX I/O service for the SBCs. The host also loads programs into the SBCs.

Each SBC has a simple multi-tasking operating system. This is an extension of a tasking system originally developed for a database machine. [5] This operating system offers simple task management services, such as creating and destroying tasks. All tasks share the same address space. Tasks can wait for, and wake up, events; events are defined by structures in shared memory. Tasks can also get and release locks; as with events, a lock is defined by a structure in shared memory.

Concurrent C is implemented on top of the SBC operating system. Unlike the UNIX implementations, in the shared memory multiprocessor, each Concurrent C process is implemented as a task in the underlying operating system. As a result the context switching overhead is slightly higher than in the uniprocessor UNIX case. However, the SBC operating system is very simple, and its context switch time is substantially less than that of a full UNIX system. The advantage is that this scheme allows Concurrent C processes to use the underlying SBC operating system facilities in a natural fashion.

UNIPROCESSOR IMPLEMENTATION: BASIC DESIGN DECISIONS

Translating to C

Our first decision was fundamental: we used C as our target language. The Concurrent C translator, ccpp (Concurrent C Preprocessor), translates Concurrent C to C. The resulting C program is then compiled and linked together with the

Concurrent C run-time library and the system libraries (Figure 3). Thus most of the machine dependencies are pushed into the local C compiler. The Concurrent C library has some machine dependencies, primarily for process switching. The Concurrent C library also depends on the multiprocessor architecture; the UNIX uniprocessor and LAN and SM multiprocessor versions of Concurrent C each have their own library. However, ccpp is independent of the machine type, and is only weakly dependent on the architecture. We have one version of ccpp for all three architectures. It generates exactly the same C code for the uniprocessor and SM multiprocessor implementations, and (using a compile-time flag) it generates only slightly different code for the LAN multiprocessor implementation.

ccpp converts some Concurrent C facilities into calls to Concurrent C run-time library functions, and converts other facilities into complex in-line code.

ccpp is a modified version of the cfront translator for C++. * Starting with cfront was much simpler than building a compiler from scratch. This also made it easy to use the same translator for both Concurrent C and Concurrent C++.

Process control blocks, process ids, transaction ids

A process control block is allocated for each process. This structure stores the process state and registers, pointers to its parent, sibling and child processes, information about the pending transaction call made by this process (if any), a pointer to the list of pending transaction calls directed to this process, transaction calls this process is ready to accept (if it is waiting at a select or an accept statement), etc. The process control blocks are elements of a global array called the process table.

A process id is a 32-bit word containing the index of the process' entry in the process table and a check count. In the multiprocessor implementations, the process id also contains the processor number. Check counts are used to detect dangling references to terminated processes. Specifically, when a process is created, a global integer is incremented, and the lower eight bits are used as the check count for that process. This check count is stored in the process id and in the process control block.

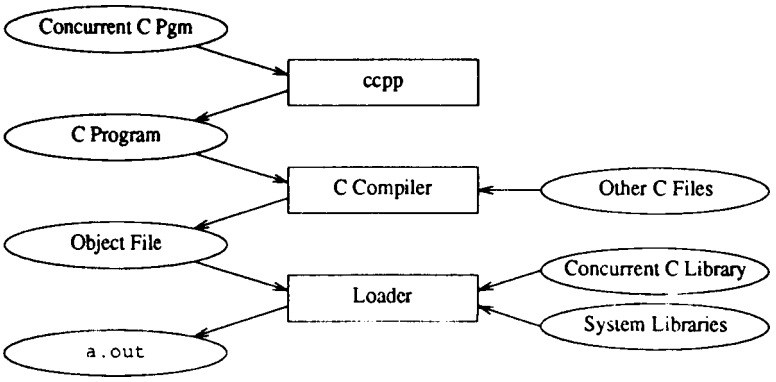

Figure 3. Compiling a Concurrent C program

* We are grateful to B. Stroustrup's help for supplying us with the cfront source

If the check count in a process id does not match that in the process control block, then the process id is a dangling reference to a terminated process.

ccpp assigns a number to each transaction listed in the process specification. The current implementation restricts a process to have at most 32 transactions. Thus when a process is waiting for several transactions at a select statement, we can represent the set of transactions with bit flags in a 32-bit word in the process control block .

Process scheduling and switching

Concurrent C processes are inexpensive 'lightweight' processes, as compared to 'heavyweight' UNIX processes. We used lightweight processes so that programmers would not hesitate, for reasons of efficiency, to use processes when appropriate. Concurrent C processes are implemented within one UNIX process. The UNIX system does not know about Concurrent C processes. Concurrent C process switching is done at the user level, and does *not* require a UNIX system call. A process switch just involves saving and restoring the machine registers.

The run-time library has a simple priority scheduler, which is invoked by a function call. That function searches the process table for a ready process, saves the registers of the old process, and resumes the new process. The scheduler function returns only when the process calling it is resumed.

Process creation and initialization, and context switching are performed by two functions. The function c_rsainit allocates and initializes the stack when a Concurrent C process is created. The function c_rsaswap switches context from the current process to another process specified by the argument. The machine-dependent process switch mechanics are isolated in these two functions; when porting to another computer, only these functions need to be changed. c_rsaswap is typically about 15 lines of assembly code; it saves the caller's registers in the old process control block, reloads the registers from the new block, and 'returns' from a c_rsaswap call in the new process. c_rsainit is about one page of C code. Arguments to c_rsainit are the address of the function that the new process is to execute, and the arguments to be given to that function. When c_rsaswap resumes the new process, it will be at a call to that function with those arguments.

A process waits for an event by changing its state to something other than 'ready', and then calling the scheduler. Another process wakes the waiting process up by changing its state to 'ready', and the scheduler eventually resumes it. We do context switches between Concurrent C processes at Concurrent C process interaction points and at time slicing intervals.

To simplify the scheduler, the run-time library starts a low priority process, the null process, which is always ready. The null process handles termination conditions, detects deadlock, and waits for pending I/O operations. Thus the scheduler always finds a ready process.

Timing: delay, alarms, and time-slicing

A Concurrent C program periodically gets an 'alarm' signal—typically every second. A function in the run-time library handles this alarm, and increments a global variable representing the current time. The run-time library has internal

facilities for suspending a process until the current time reaches a particular value. The alarm handler wakes up processes at the desired time. We use this to implement the Concurrent C delay statement and other other time-oriented facilities.

To prevent one process from monopolizing the machine, the alarm handler forces a process switch when the alarm arrives, unless the alarm occurs in a critical section (see below). That is, the alarm handler asks the Concurrent C scheduler to run another ready process of the same or higher priority. Although this is not true time-slicing, it has low overhead, and it ensures that a CPU-bound Concurrent C process will not prevent other processes of the same or higher priority from executing.

Critical sections

A critical section is a piece of code that must not be interrupted by a process switch. The Concurrent C run-time library functions, which manipulate internal data structures, must not be interrupted because this may leave data structures in inconsistent states. Many C (and C++) library functions should also not be interrupted; for example, the storage allocation function, malloc, must not be interrupted while it is updating its tables. An alarm signal can interrupt execution of a critical section; however, if the alarm handler detects that the interrupt process was in a critical section, then it always resumes that process; it will not switch to another process.

We use two mechanisms to implement critical sections. The first is for critical sections within the Concurrent C library. It uses a global variable, which is initially zero. Each critical section increments that variable on entry, and decrements it on exit. Thus the alarm handler does not force a process switch if that variable is non-zero. A counter simplifies nesting of critical sections.

The second critical section mechanism is for library functions such as malloc. We could have provided 'atomic' versions of those library functions, but it would be a nightmare to maintain all these different copies. Instead, we instruct the loader to put the Concurrent C program and run-time library in lower addresses, and put the system library in higher addresses. The alarm handler examines the program counter at the time of the alarm. If the program is interrupted in the system library portion of the address space, then the alarm handler does not force a process switch. This scheme has the advantage of being safe, easy to implement and maintain, and reasonably portable. The disadvantage is that every system library function becomes a critical section; if a CPU-bound Concurrent C process spends all its time in such functions, it will keep other processes from executing. So far this has not been a problem.

Reading from slow devices

The UNIX read system call is synchronous: it waits until the read completes. Because UNIX does not know about Concurrent C processes, a read system call blocks all processes in a Concurrent C program until the read completes. That is acceptable for reading from a fast device such as a disk, but when reading from a slow device such as a terminal, we would like other Concurrent C process to be able to run while the requesting Concurrent C process is waiting for the read to complete.

To accomplish this, we supply our own version of the read system call. On the

Berkeley UNIX system [6] our version uses the select system call to determine if the operation can be done immediately. If yes, read does the system call and returns; otherwise, read puts the calling Concurrent C process into a 'waiting for I/O' state, saves the file descriptor in the process control block, and calls the scheduler to run another process.

When the scheduler encounters a process that is waiting for I/O, the scheduler uses the select system call to determine whether or not the device is ready. If yes, the scheduler marks this process as ready, and resumes it so that it can do the operation and return to the caller.

Null process

The null process is started by the Concurrent C run-time system. It is guaranteed to have a lower priority than any user process, and it therefore runs only when no other processes are ready. The null process handles termination conditions and detects deadlock. When it runs, the null process scans the process control blocks to determine the state of all processes. If all processes are waiting at a select statement with a terminate alternative, then program termination conditions have been satisfied, and the null process stops the program. Otherwise, if some processes are waiting for I/O or for a delay to expire, then the null process waits for the next alarm signal or for any of the awaited devices to become ready, and asks the scheduler to pick another process. If none of these conditions is satisfied, we have deadlock: the null process prints an error message and stops the program.

PROCESS STATES

During execution, each Concurrent C process is in one of eight internal states. Figure 4 shows the state transitions.

The c_ready state identifies processes that are ready for execution and waiting to be scheduled. When picking the next process to run, the scheduler selects the highest priority ready process. All processes are created in the c_ready state. From the c_ready state, processes move to the c_current state when they are selected for execution.

The currently executing process is in the c_current state. From this state, a process moves to the

(a) c_ready state if the process is rescheduled as a result of a timer interrupt
(b) c_complete state if it completes execution of its body
(c) c_wservice state if it issues a transaction call
(d) c_wselect state if it must wait at an accept or a select statement
(e) c_wread state if it must wait to read from a 'slow' device, such as a terminal.

The c_wservice ('waiting for service') state indicates that the process has issued a transaction request that has not yet been accepted by the called process. Processes in this state wait in a queue associated with the called process. For a timed transaction call, if the called process does not accept the call within the time limit, then the call is withdrawn and the calling process enters the c_ready state. A process moves to the c_trans state when its transaction call is accepted.

The c_trans ('in transaction') state indicates that the synchronous transaction call

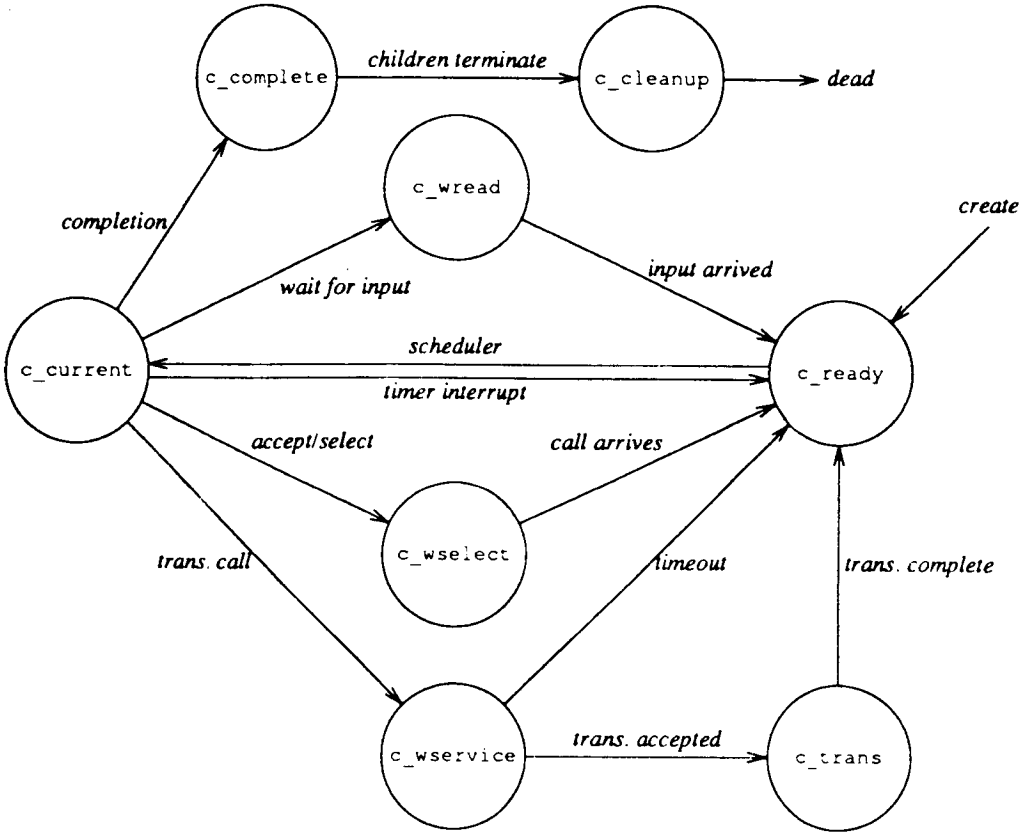

Figure 4. State transitions

issued by a process has been accepted by the called process, and the called process is executing the body of the accept statement (which is used to accept transactions). A process moves to the c_ready state when the called process completes the transaction.

The c_wselect state indicates that a process is waiting at a select or an accept statement. A process reaches this state when it executes one of these statements and finds that it must wait for (a) a transaction call request for some subset of the process' transaction types, (b) a time delay or (c) termination conditions. A process moves to the c_ready state when a suitable transaction call arrives or when the delay expires.

A process reaches the c_complete state by reaching the end of its body. It waits in the c_complete state until its last child terminates, and then it moves to the c_cleanup state.

The c_cleanup state is similar to the c_ready state in that the process is ready to run and can be selected by the scheduler. However, in the c_cleanup state, a process executes only the run-time library functions that perform system clean-up actions and then terminates the process. The scheduler gives c_cleanup processes priority over c_ready processes.

A process reaches the c_wread ('waiting for read') state when it issues a read

system call on a slow device (e. g., a terminal), and the read cannot complete immediately. The process moves to the c_ready state when data is available.

TRANSACTION CALLS

There are three types of transaction calls: synchronous (unconditional), timed synchronous and asynchronous. Similar techniques are used to implement all three types. The process specification tells whether a transaction is synchronous or asynchronous. By definition, an asynchronous transaction cannot be timed and cannot return a value (it is void -valued). A synchronous transaction call can be timed, at the caller's option. Thus the person who writes the process specification decides whether or not a transaction is asynchronous, while the person who writes the transaction call decides whether or not a timeout is associated with a particular transaction call.

For each transaction listed in a process specification, ccpp generates an interface function to handle the mechanics of the call. The interface function has the same return type as the transaction, and takes the same arguments, plus the process id of the called process. ccpp replaces each transaction call with a call to the corresponding interface function. In addition, ccpp generates a structure whose components correspond to the transaction arguments. This structure gathers the arguments into a contiguous area of memory.

Here is a quick summary of the transaction call mechanics (details are given later):

1. When a process issues a transaction call, the interface function constructs a call-description structure, adds that structure to the tail of the pending call list for the called process, wakes up the called process if it is waiting for such a call, and waits for the call to complete.
2. When the called process accepts the call, it removes the call-description structure from its list and executes the body of the accept statement.
3. When the called Process completes the call, it copies the return value into the indicated area and wakes up the calling process.

Data structures

The call description structure, of type c_tcall, contains all the information about one pending transaction call. It contains the bit-mask for the transaction (corresponding to its number). It also points to the arguments for this call, the area into which the called process is to put the returned value, a pointer to the process control block (c_proc) for the calling process, and the next c_tcall for the called process:

```
typedef struct C_TCALL c_tcall;
struct C_TCALL {
    c_tcall    *tc_next;        /* next pending call */
    c_proc     *tc_caller;      /* PCB of calling process */
    long       tc_msk;          /* 1<< trans call # */
    char       *tc_args;        /* arg list */
    char       *tc_ret;         /* where to put treturn value */
};
```

As illustrated in Figure 5, component p_tcin of the process control block of the called ('server') process points to the head of the list of c_tcall structures for the pending transaction calls directed to this process.

A process has one list for all the pending transaction calls instead of one list each for each transaction type. A list for each transaction type would make it easier to find the next transaction of a given type. However, most processes use a select statement to wait for one of several types of transactions. It is easier to implement this with a single transaction list. Furthermore, a list for each different transaction type would require 32 pointers in the process control block (a process can have up to 32 transaction types), and that would double the size of the block.

The transaction lists are singly linked. A doubly linked list would make it faster to add an entry at the end of the list or to remove one from the middle. However, in most cases the list is very short (usually one pending call). Thus the trade-off is the overhead of maintaining doubly linked links for many short lists versus the overhead of linear search on the occasional long list. We choose simplicity.

Synchronous transactions

A transaction call is converted to a call to the interface function whose name has the form c_TC ⟨ *tnum* ⟩⟨ *pname* ⟩, where *tnum* is the number assigned internally to the transaction, and *pname* is the process name. The name of the structure containing the transaction arguments has the form c_TS ⟨ *tnum* ⟩⟨*pname* ⟩ As an example, consider the process specification:

```
process spec foo()
    { trans int bar(int a, double b); };
```

ccpp assigns the transaction number 0 to bar, and generates the following declarations:

```
/* foo,bar (): transaction 0 */
int c_TC0foo ();          /* interface function, unconditional calls */
typedef struct {          /* argument structure */
    int       c_TS0_0foo; /*arg 0*/
    double c_TS1_0foo; /*arg 1*/
} c_TS0foo;
```

ccpp translates the transaction call p. bar(a, b) into the interface function call c_TC0foo(p,

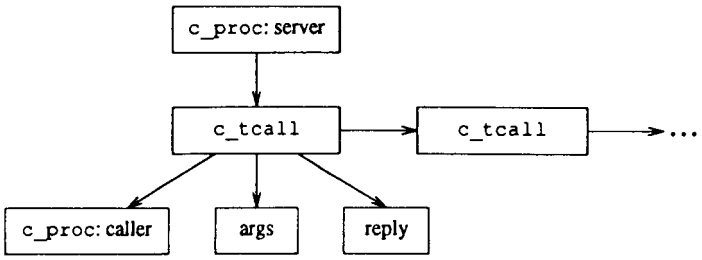

Figure 5. Pending transaction call descriptors

a, b). ccpp generates the intertface function bodies when it encounters the process body. Each interface function has local variables for the argument structure and for the return value. The interface function copies its arguments into the argument structure, calls the common run-time function c_tc to actually make the transaction call, and finally returns the result to the caller. Here is the body of the interface function

```
c_TC0foo:
    int    c_TC0foo  (pid, p0, p1)
           c_pid      pid;
           int        p0;
           double     p1 ;
    {      int        result;
           c_TS0foo   args;

           args.c_TS0_0foo = p0;
           args.c_TSl_0foo = p1;
           c_tc (pid, 1<<0, (char*) &result, (char*) &args);
           return result;
    }
```

The arguments to c_tc are the id of the called process, the bit-mask for the transaction number, and pointers to the argument and reply areas. c_tc uses a c_tcall structure in the caller's process control block to store information about a pending transaction. Here is an outline of what c_tc does:

1. Start a critical section.
2. Verify that the called process id—the server—is valid.
3. Setup the c_tcall structure in the caller's process control block
4. Add the c_tcall structure to the end of the server's list.
5. Wake up the server if it is waiting for this transaction type.
6. Set the caller's state to c_wservice.
7. End the critical section.
8. Call the scheduler to pick another process to run.

Because the calling process is no longer ready, the scheduler will always pick another process (usually the server process). When the server completes the transaction, it copies the return value into the caller's area, and changes the caller's state to c_ready. Eventually the scheduler resumes the calling process, and c_tc returns to the interface function.

The called process is waiting for a transaction if it is in state c_wselect, and if the bit for the number assigned to this transaction is on in the bit-mask representing the transactions the called process is waiting for. To wake up the called process, c_tc just changes its state to c_ready.

Note that we do not explicitly allocate or free memory or message buffers for a synchronous transaction call. The c_tcall is pre-allocated, because it is in the caller's process control block. The arguments and reply area are on the caller's stack, and are implicity allocated by calling the interface function, and freed when it returns.

Timed and asynchronous transaction calls

For each synchronous transaction, ccpp generates an additional interface function to handle timed calls. This is similar to the one for unconditional calls. The difference is that the timed call function asks to be put in the ready state when either the time delay expires or the server completes the transaction. When the function is resumed, it determines whether or not the call completed. If so, it returns the value from the server. If not, it atomically withdraws the pending call request.

For each asynchronous transaction, ccpp generates only one interface function (asynchronous transaction calls cannot be timed). It is similar to that for synchronous transactions, with two major differences. The first is that it allocates a c_tcall structure instead of using the one in the caller's process control block, and allocates an argument structure instead of using an automatic variable. The other is that after adding the c_tcall to the server's list, the interface function returns immediately to the caller. When the server completes the asynchronous transaction, the server frees the structures allocated by the interface function.

ACCEPTING TRANSACTION CALLS

Simple accept statements

ccpp translates an accept statement into a block of C code. The block has local pointer variables to the c_tcall structure, the argument area, and the reply area for this transaction. The block has a prologue, which gets the transaction call, a body, which is derived from the body of the accept statement, and an epilogue, which completes the call. The block body is identical to the body of the accept statement, with three exceptions. First, each reference to a transaction parameter is replaced with a reference to the corresponding component in the transaction argument structure. Secondly, each treturn statement is replaced by a statement to copy the result into the return value area, and a branch to the epilogue. Finally, for any branch outside of the accept body, a copy of the epilogue code is generated before the branch. As an example, consider an accept statement for a synchronous transaction bar for the process type foo:

```
accept bar(a,b) { treturn myfun(a, b); }
```

Here is the C code generated for the above accept statement; we have added white space and comments for readability. We assume that the C compiler cleans up inefficient code, such as the branch to the next statement in this example. This assumption allows us to simplify ccpp.

```
{       register c_tcall      *c_caller0;    /* for call */
        register c_TS0foo     *c_AA0;        /* → arguments */
        int                   *c_result;     /* → reply area */

    /* prologue: get transaction call */
        c_caller0 = c_waccept (1 <<0); /* bar: trans num 0 */
        c_AA0 = (c_TS0foo* ) c_caller0 → tc_args;
```

```
        c_result = (int*) c_caller0 → tc_ret;

    /* accept body */
        *c_result = myfun (c_AA0 → c_TS0-0foo, c_AA0 → c_TS 1 _0foo);
        goto c_Acc0;

    /* epilogue: complete transaction */
    c_Acc0: c_treturn (c_caller0);
}
```

The 0 suffix in c_caller0 and c_AA0 is a sequence number for accept statements in this process, and ensures that the variable names used are unique. This is needed because accept statements can nest, and an inner accept statement can refer to parameters of an outer accept statement.

Function c_waccept dequeues and returns the c_tcall structure for a pending transaction call for the process calling the function. The argument is a bit-mask for the desired transaction type. c_waccept scans the pending call list of this process for the first entry with the correct transaction type. If there are no such entries, c_waccept waits until one arrives. When a call is available, c_waccept changes the state of the calling process to c_trans, removes the c_tcall from the list, and returns it:

```
c_tcall *c_waccept (tmask)
    int tmask;
{ start critical section;
    while (1) {
        if ( pending call list does not have an entry of type tmask) {
            save tmask value in process contro/ block;
            set process state to c_wselect;
            end critical section,
            call scheduler;
            start critical section,
        }
    }
    remove entry from queue;
    change state of calling process to c_trans;
    end critical section;
    return pointer to dequeued c_tcall entry;
}
```

The c_treturn function wakes up the calling process by changing its state to c_ready. The return value has already been set, so that is all that is needed. Note that we use the arguments ‘in-place’; we do not copy them.

The same code works for timed transaction calls as well as for synchronous transaction calls. If the timeout expires before the called process accepts the call, the calling process atomically removes the request and the called process never sees it. If the timeout expires after the called process has accepted the call, the call will complete; it cannot be withdrawn. To ensure this, when the alarm handler sees that a timeout has expired, if the process is in the c_trans state, the alarm handler ignores the timeout instead of readying the process.

If the transaction is asynchronous, the epilogue calls the library function c_treturna to free the c_tcall structure and the argument area which the caller had allocated.

Accept statements with by or suchthat clauses

By default, transaction calls are accepted in FIFO order. The by clause can be used to reorder these calls and the suchthat clause can be used to screen out unwanted calls. These clauses allow examination of the transaction call parameters without accepting any of the calls except the desired one.

For an accept statement with a by or a suchthat clause, the generated prologue is more complicated. Instead of calling a function in the run-time library to scan the list of pending calls, ccpp generates the scan-loop in-line. If there is a suchthat clause, ccpp generates code to evaluate the suchthat predicate for each pending call of the desired type, and skips those for which the predicate is false. If there is a by clause, ccpp generates code to scan the entire list to find the pending call with the lowest value of the by expression, and then take that call.

Select statements

First, a quick review. A select statement has a set of (possibly guarded) alternatives. There are four types of alternatives: accept, immediate, delay and terminate. One of the alternatives is executed; it is selected as follows:

1. If there is a transaction call pending for one of the accept alternatives, take that call, assuming that the guard for the alternative is open, and the suchthat clause, if any, is satisfied.
2. Otherwise, if there is an immediate alternative with an open guard, take it.
3. Otherwise, wait for a transaction call that can be accepted, the expiration of the delay, or the occurrence of the termination conditions.

Termination conditions are recognized by the low-priority null process. This process terminates the program normally if it sees that all Concurrent C processes are waiting at a select statement with an open terminate alternative. Therefore we just set a flag in the process control block to indicate whether or not the select statement has a terminate alternative.

As an example, consider the select statement in the process shown below. * The guards are Boolean expressions; in this case, they are calls to the functions g0, g1, etc.

```
process spec foo() {
    trans void bar0 (int a, double b);
    trans void bar1 (int a, double b);
};

process body foo() {
    select {
        (g0()) : accept bar0(a,b) {f0(a,b); }
```

* This select statement is not supposed to make sense; it simply illustrates the various kinds of alternatives

```
        or
            (g1 ()) : accept bar1 (a,b) suchthat (a= =5)    {f1 (a,b); }
        or
            (g2()) : f2();
        or
            (g3()) : delay 4.0; f3();
        or
            (g4()) : terminate;

    }
```

For this select statement, ccpp generates a block of C code with declarations, a prologue, and the code for the body of each alternative. Figure 6 gives the complete C code; here is an outline:

```
{ delarations
    prologue: select alternative and branch to it; wait if necessary
    body of first (accept) alternative
    body of second (accept) alternative
    body of third (immediate) alternative
    bodies of all delay alternatives
}
```

Here is a skeleton of the prologue:

```
evaluate guards of accept alternatives, determine set of transaction types;
while (1) {
    scan pending calls; if any are acceptable, goto that accept alternative;
    if (first time) {
        evaluate guards for immediate alternatives; if any are open, goto that
        alternative;
        evaluate guards for terminate and delay alternatives;
    }
    wait for event, transaction call, delay, or terminate;
    if (delay has expired)
        goto delay alternative;
}
```

When an alternative can be taken, the prologue branches to that alternative. If not, it waits and tries again. The loop in the prologue has three parts. The first starts a critical section and scans the pending transaction call list. Macros c_DISABLE and c_ENABLE start and end a critical section (they ‘disable’ and ‘enable’ interrupts). The global variable c_cur points to the process control block of the currently running process. If a pending transaction is acceptable—it has the correct type, the guard is open, and it satisfies the suchthat clause—the prologue sets c_p to point to that call, and branches to the corresponding accept body.

If there are no suitable pending transaction calls, the second part of the prologue evaluates the guards on the terminate, immediate and delay alternatives, if it has not

```
{   int     c_atdel - 0,  c atterm = 0;   /* true if have delay or term alts */
    double  c_del;
    l o n g   c_tmask = 0;   /* trans type bit-mask */
    long    c_gmask - 0;   /* open–alt bit–mask, for accepts */
    c_tcall *c_p;
    int     c delalt, c first;

  /* evaluate accept guards, get bit-mask for desired trans numbers */
    if (g0()) ( c_gmask 1= (1<<0); c_tmask |= (1<<0); } /* trans number 0 */
    if (g1()) { c_gmask 1= (1<<1); c_tmask |= (1<<1); } /* trans number 1 */

  /* prologue:  select an alternative and branch to it */
    for (c_first = 1;  1; c_first = 0) {

      /* scan pending calls, branch if found */
        c_DISABLE;    /* start critical section (macro) */
        for (c_p = c_cur->p_tcin;  c>;  c_p  =  c_p->tc_next)  {
            switch (c_p->tc_msk) (
            case 1: /* bar0 */
                if (c_gmask & 1) goto c_SellAccO;
                break;
            case 2: /* bar1 */
                if (c_gmask & 2) {
                  /* check suchthat predicate */
                    register  c_TS1foo*c_AA1:
                    c_AA1 - (c_TSlfoo*) c_p->tc_args;
                    if (c_AA1->c_TS0_1foo == 5) goto c_SellAcc1;
                }
                break;
            }
        }
      /* if first time,  evaluate  immediate/delay/terminate  guards  */
        if (c_first) (
          /* goto any open immediate alt */
            if (g2()) ( c_ENABLE;   goto c_Sel1Imm2; )
          /* set c atterm if any terminate alts are open */
            if (g4()) c atterm = 1;
          /* find lowest delay value and alt number */
            if (g3()) {
                double c_deltmp = 4.0;
                if (!c_atdel || c_deltmp < c_del) {
                    c atdel = 1;  c delalt = 3;  c_del = c_deltmp;
                }
```

Figure 6. C Code' generated for select statement

already done so. If there is an open immediate alternative, it branches to the code for it. Otherwise c_atterm and c_atdel are set to l if we have either an open terminate or delay alternative. In general there can be several delay alternatives. The prologue sets c_del to the minimum delay value and c_delalt to the number of that alternative.

The last part of the prologue loop calls the function c_evwait to wait for the first of the selected set of events. Function c_evwait does the mechanics of waiting for a transaction call or some other event to occur. The first argument is a bit-mask specifying the transaction for which the process is waiting. The second argument is

```
        }
      /* wait for an event */
        if (c_TIMEOUT -- c_evwait (c_tmask, c_atterm, c_atdel, &c_del) )
            goto  c_Sel1Del;
    }
  c_Sel1Acc0: /* first alternative: accept foo.bar0() */
    {   register c_tcall   *c_caller0 - c_p;
        register c_TS0foo *c_AA0:
        c_accept (c_caller0) ;
        c AA0 - (c_TS0foo*) c_caller0->tc_args;
        c_ENABLE ;    /* end critical section */
        f0(c_AA0->c_TS0_0foo,  c_AA0->c_TSl_0foo);
        c_treturn (c_caller0) ;
    }
    goto c_Sel1;
  c sel1Acc0: /* second alternative: accept foo.bar1() */
   { similar to code for first alternative }
    goto c_Sel1;
  c_Sel1Imm2: /* third (immediate) alternative: f2() */
    f2(); goto c_Sel1;
  c_Sel1Del:  /* fourth (delay) alternative: f3() */
    if (c_delalt == 3) ( f3(); goto c_Sel1; )
  c_Sel1:;
}
```

Figure 6. Continued

1 if there is an open terminate alternative, and the third arguments 1 if there is a delay alternative, in which case the last argument points to the delay value. Function c_evwait changes the state of the calling process to c_wselect, exits the critical section, and calls the scheduler to pick another process. When c_evwait returns, it is *not* in a critical section.

If the termination conditions occur, c_evwait does not return, because the null process kills the entire program. Otherwise the return code from c_evwait tells whether a transaction arrived or the delay expired. If the delay expired, the prologue branches to the code for the delay alternatives. If a call arrived, the prologue loops back and scans the list again. If the call satisfies the appropriate suchthat clause, the prologue accepts that call. If not, it waits again.

If c_evwait returns because a call has arrived, it sets the last argument to the remaining delay time. If the new call is not acceptable and c_evwait is called again, it will wait for the remaining delay. Thus the total delay will be from when the select statement started, and not from the last time the process woke up to check the list. This method does introduce a small error in the delay, because it does not count the time that the process spends waiting to be rescheduled after the call arrived, or the time to check the call list. However, in most cases this is a small error, and given that on most systems the time resolution is one second, it makes little difference.

Because of suchthat clauses, a process may wake up and scan the pending-call list several times before it gets an acceptable call. However, the process does *not* 'poll'

or 'busy-loop'. As an optimization, we could just examine the calls that have been added since the last time the process scanned the list of transactions. For example, we could keep a pointer to the 'last checked' call, and start from there when the process is re-awakened. We have not done this because usually the list is short, and timed calls can be withdrawn by the caller, which makes it difficult to keep a pointer to the 'last checked' call.

The code for each accept alternative has a prologue, body, and epilogue. The body and epilogue are identical to those generated for a stand-alone accept statement. The prologue differs. At entry, we are in a critical section, and the local variable c_p points to a satisfactory c_tcall structure. The accept statement prologue dequeues that c_tcall and exits the critical section. Function c_accept does the common mechanics of accepting a transaction call: it dequeues the c_tcall structure passed as its argument, changes the caller's state to c_trans, etc.

The suchthat clause (if any) is handled in the prologue of the select statement. If the accept alternative has a by clause, ccpp generates code in the prologue of the accept alternative to scan the list of pending calls and pick the one with the lowest value.

An alternative to generating the scan-loop in-line would be to generate a function for each suchthat and by clause, and pass pointers to those functions to a common search function. The generated C code would be smaller and simpler, ccpp would be simpler and easier to change because it would have less knowledge of process control blocks, etc. However, these clauses can (and usually do) reference local variables of the process containing the select statement. With nested select statements, they can even reference parameters of an enclosing select statement. The C code to make these variables available to separate functions would be much uglier than the in-line code.

SUMMARY

We simplified the task of implementing a Concurrent C compiler by translating Concurrent C and Concurrent C++ to C and C++. All C statements in a Concurrent C program are output essentially without any transformation. We did not have to worry about the C or C++ statements because we started with a C++ translator which already took care of them.

In our UNIX implementation, we have implemented lightweight processes which run within a UNIX process. The UNIX system does not know anything about these lightweight processes. As far as the UNIX system is concerned, the Concurrent C program is an ordinary sequential program. We do our own context switching and we provide our own scheduler which runs separately from the UNIX scheduler.

We are currently in the process of developing a new version of our compiler. Previously, ccpp translated Concurrent C++ directly to C, bypassing (and replacing) the C++ compiler. We did that because ccpp was a modified version of the C++ translator, so it had all the necessary C++ translation algorithms. However, when those C++ translation algorithms changed, we were not able to keep ccpp up-to-date with the C++ compiler. The result was that Concurrent C++ object code became incompatible with C++ object code, and we had to maintain separate copies of the C++ libraries. To avoid these problems, we recently modified ccpp to transform Concurrent C++ code into C++, which it then gives to the local C++

compiler. That ensures that Concurrent C++ is compatible with C++, and makes it easy to pick up advances in C++ compiler technology.

We have given you many details of the Concurrent C implementation. By doing this, we hope that we have taken some mystery out of how concurrent programming facilities can be implemented.

ACKNOWLEDGEMENTS

Bob Cmelik implemented release 1.0 of the Concurrent C translator (ccpp) and the run-time library and helped develop many of the code translation techniques. Currently, Jacques Gava is working on release 2.0 of Concurrent C. Bjarne Stroustrup provided the cfront source, which was the basis for ccpp. Finally, we are grateful to the anonymous referees for their many comments and suggestions.

REFERENCES

1. N. H. Gehani and W. D. Roome, 'Concurrent C++: concurrent programming with class(es)', *Software—Practice and Experience*, **18.** 1157–1177 (1988).
2. N. Gehani and W. D. Roome, in *The Concurrent C Programming Language*, Silicon Press, 1989.
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
4. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
5. W. D. Roome and M. D. P. Leland, The silicon database machine, *Proc. 5th Int. Workshop on Database Machines, 1985*, pp 169–189.
6. *UNIX Programmer's Manual (4.2 BSD)*, Computer Science Division, EECS, University of California, Berkeley, 1983.