Strata: A Multi-Layer Communications Library
Version 2.0 Beta

Eric A. Brewer and Robert Blumofe*
MIT Laboratory for Computer Science

February 15, 1994

Strata is a multi-layer communications library under development at MIT. Version 2.0 provides safe, high-performance access to the control network and flexible, high-performance access to the data network.

Strata works correctly on top of CMMD 3.0; both the data- and control-network operations can be mixed with Strata calls. However, Strata forms a complete high-performance communication system by itself, with substantial support for degugging and monitoring.

Strata provides several advantages over CMMD 3.0 [TMC93]:

Support for split-phase control-network operations. Split-phase operations allow the processor to perform work while the global operation completes. Strata provides split-phase barriers, segmented scans, and reductions.

Support for debugging and monitoring. Strata provides versions of `printf` that do not poll and can be used within handlers. This allows users to insert print statements for debugging without changing the atomicity of the surrounding code. Strata also provides routines for accurately timing short events, and for clean assertion checking and error handling. Finally, Strata provides support for automatic generation of state graphs, which depict the state of each processor versus time.

Support for developing high-performance data-communication protocols. Strata provides a rich and flexible set of data-network procedures that can be used to implement complex data-communication protocols. By understanding certain potential pitfalls, the Strata user can implement protocols that are more complex, are more robust, and achieve higher performance than those that can be implemented with the procedures and usage-rules of CMMD 3.0.

Higher performance. Table 1 compares the performance of Strata with that of CMMD 3.0. Strata's control-network procedures are considerably faster than CMMD's (though these operations rarely dominate execution time).

---

*E-mail: `brewer@lcs.mit.edu` or `rdb@lcs.mit.edu`. This work is supported by Project SCOUT (ARPA Contract MDA972-92-J-1032); by the National Science Foundation, grant CCR-8716884; by ARPA, Contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM. Eric Brewer received support from an Office of Naval Research Fellowship. Robert Blumofe is supported by an ARPA High-Performance Computing Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

CMMD 3.0		Strata	
Primitive	**Time**	**Primitive**	**Time**
`CMMD_sync_with_nodes`	170 cycles	`Barrier`	140 cycles
`CMMD_scan_int`	380 cycles	`CombineInt`	150 cycles
`CMMD_scan_v`	241 cycles/word	`CombineVector`	49 cycles/word
`CMMD Broadcast, 1 word`	230 cycles	`Broadcast`	90 cycles
`CMMD Broadcast, double`	380 cycles	`BroadcastDouble`	90 cycles
`CMMD Broadcast, vector`	150 cycles/word	`BroadcastVector`	49 cycles/word
`CMAML_request`	67 cycles	`SendLeftPollBoth`	43 cycles
`CMAML_poll`	45 cycles	`PollBothTilEmpty`	22 cycles
`bzero`	36 cycles/word	`ClearMemory`	21 cycles/word

Table 1: Relative performance of Strata. The cycles counts for the active message procedures (`CMAML_request` and `SendLeftPollBoth`) as well as for the polling procedures (`CMAML_poll` and `PollBothTilEmpty`) are for the case when no messages arrive.

Contents

1 Using Strata

Strata programs have their own default host program and are linked with the Strata library. The Strata directory includes an example program (radix sort) that uses many of the Strata procedures.[1] Primary files:

File	Description
README	Installation notes
INSTALL	Installation script
strata.h	Defines all of the prototypes, typedefs and macros
libstrata.a	The Strata library, linked in with -lstrata
libstrataD.a	The Strata debugging library, linked in with -lstrataD
strata_host.o	The default host program object file
Makefile	The make file for the radix-sort application
radix.c	Radix-sort source code
do-radix	The DJM script for running radix sort
control_net_inlines.c	Inline control-net procedures, included by strata.h
data_net_inlines.c	Inline data-net procedures, included by strata.h
strata_inlines.c	Basic inline procedures, included by strata.h
src	Directory containing the Strata source code
strata.ps	The PostScript for this document.

See README for installation notes. After installation, users should be able to copy Makefile, radix.c, and do-radix into their own directory and build the example application. Makefile contains the necessary options to include strata.h and to link with the Strata library.

2 Strata Functionality

The rest of this document covers the current set of Strata routines. Figure 1 shows the structure of the Strata layering; each block corresponds to one of the following sections. Appendix A lists the prototypes for reference. There are several groups of Strata routines:

[1] For MIT Scout users, the Strata library and include files are currently in /u/brewer/strata.

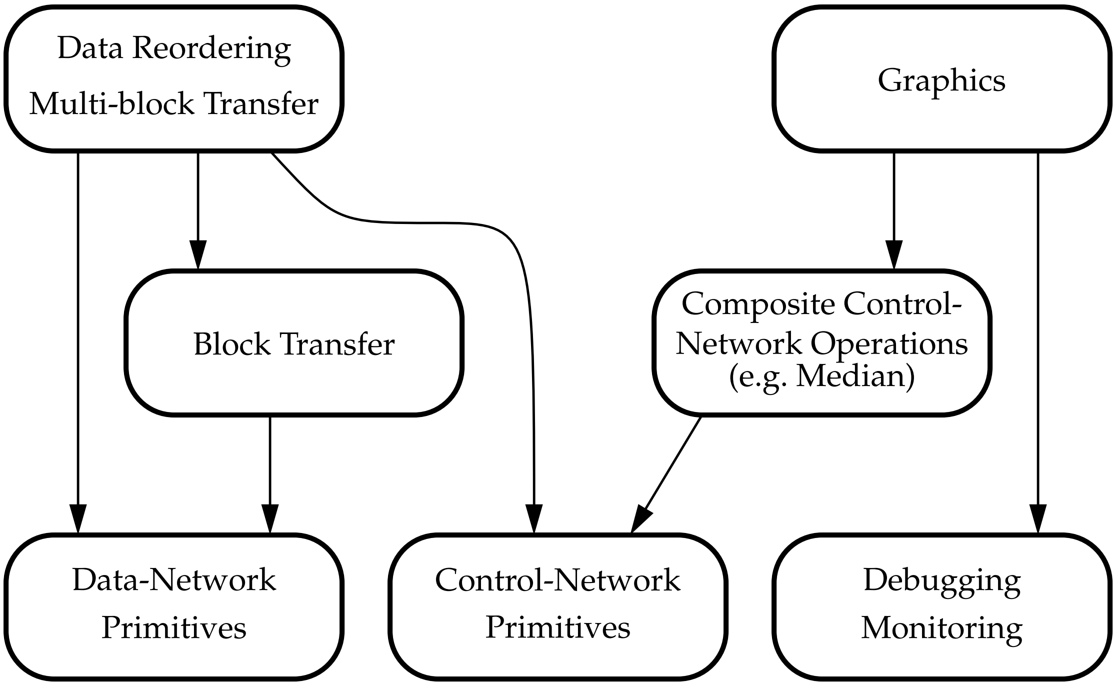

Figure 1: The layering of Strata.

	Section	**Description**
3	Basics	Initialization, error handling, basic macros and globals
4	Control Network	Primitive control-network operations
5	Composite Reductions	Higher-level reductions
6	Data Network	Sending and receiving short data messages
7	Block Transfers	Sending and receiving blocks of data
8	Multi-Block Transfer	Higher-level data movement
9	Debugging	Support for debugging and timing
10	Graphics	Support for state graphs

3 Strata Basics

3.1 Type Qualifiers

`ATOMIC`	Used in function prototypes indicate that the procedure does not poll. This in only a convention; it has no effect on the procedure.
`NORETURN`	Used in function prototypes to indicate that the procedure never returns (like `exit`).
`HANDLER`	Used in function prototypes to indicate that the procedure is a handler. This in only a convention; it has no effect on the procedure.

3.2 Globals

`int Self`	Local node number
`int PartitionSize`	Current partition size
`int logPartitionSize`	The log (base two) of the partition size
`int PartitionMask`	Mask for legal node bits
`char *StrataVersion`	String: version number and date
`float StrataVersionNumber`	The version number

3.3 Types

`Word`	A generic 32-bit quantity (unsigned)
`DoubleWord`	A generic 64-bit quantity (unsigned)
`Bool`	Used for booleans; can be set to `True` (1) or `False` (0)

3.4 Macros

`LEFTMOST`	Defined as `(Self==0)`
`RIGHTMOST`	Defined as `(Self==PartitionSize-1)`

3.5 Procedures

`void StrataInit(void)`

This procedure initializes Strata. In particular, it:

1. Enables CMMD,

2. Disables interrupts,
3. Initializes the Strata global variables,
4. Sets `stdout` and `stderr` to independent append mode, and
5. Enables broadcasting.

`NORETURN StrataExit(int code)`

Exits the application after cleaning up. It is an error to exit without going through `StrataExit`, so Strata redefines `exit` and `assert` to exit through `StrataExit`. If `code` is non-zero, the entire application will exit immediately. If it is zero, then the node idles until all processors reach `StrataExit`, at which point the entire application exits normally. During the idle time, the node handles incoming messages (by polling).

`NORETURN StrataFail(const char *fmt, ...)`

The arguments work like `printf`. After printing the message to `stderr`, it exits through `StrataExit` with error code -1.

`ATOMIC void ClearMemory(void *region, unsigned length_in_bytes)`

This procedure zeroes out the region. It is functionally equivalent to `bzero`, but is faster because it uses double-word writes exclusively. It is provided primarily as an example of how to use double-word accesses from Gnu C. (Source code is in `memory.c`.)

3.6 Random-Number Generator

`ATOMIC unsigned Random(void)`

Returns a pseudo-random 32-bit value. This is the "minimal standard" random-number generator described by Park and Miller [PM88]. The default seed is `Self`+1 (set during `StrataInit`), which gives each processor a unique but repeatable sequence.

`ATOMIC unsigned SetRandomSeed(unsigned seed)`

Sets the random seed for this node to `seed`. If `seed` is the special value `TIME_BASED_SEED`, then the seed is set based on the cycle counter and the processor number. This results in different seeds for different runs (as well as for different nodes). The return value is the seed actually used, which can be used later to repeat the sequence.

4 Control-Network Primitives

4.1 Barriers

`void Barrier(void)`

Returns when all nodes reach the barrier. It is equivalent to `GlobalOR(False)`.

`ATOMIC void StartBarrier(void)`

Starts a split-phase barrier.

`ATOMIC Bool QueryBarrier(void)`

Returns true if and only if the last barrier has completed.

`void CompleteBarrier(void)`

Returns when the last barrier has completed, possibly blocking.

4.2 Global OR

`Bool GlobalOR(Bool not_done)`

Returns the OR of the `not_done` argument of each PE. All nodes must participate.

`ATOMIC void StartGlobalOR(Bool not_done)`

Starts a global OR operations, but returns immediately.

`ATOMIC Bool QueryGlobalOR(void)`

Returns true if and only if the last global OR has completed.

`Bool CompleteGlobalOR(void)`

Returns the result of the last global OR; blocks until the result is available.

`ATOMIC void SetAsyncGlobalOR(Bool not_done)`

This is an asynchronous version of the global-OR operation. This procedure sets this processor's contribution. During `StrataInit`, each node's value is set to `True`.

`ATOMIC Bool GetAsyncGlobalOR(void)`

Reads the asynchronous global OR bit. Returns true if and only if at least one node has its asynchronous-global-OR bit set to true.

4.3 Combine Operations

The control network provides five associative functions for combine operations: signed add, or, xor, unsigned add, and signed max. Thus, there are fifteen primitive combine operations:

```
ScanAdd     BackScanAdd     ReduceAdd
ScanOr      BackScanOr      ReduceOr
ScanXor     BackScanXor     ReduceXor
ScanUadd    BackScanUadd    ReduceUadd
ScanMax     BackScanMax     ReduceMax
```

The `Scan` variants perform a forward scan (by processor number), the `BackScan` variants perform a backward scan, and the `Reduce` variants perform a reduction. Together these fifteen variants form the enumerated type `CombineOp`, which is used by all of the Strata combine primitives.

4.3.1 Segmented Scans

Strata also offers segmented scans. The segments are global state maintained by the hardware; Strata scans implicitly use the current segments. Initially, the partition is set up as one large segment covering all nodes. Note that reduction operations ignore the segments (this is a property of the hardware), but segmented reductions can be built out of two segmented scans. Normally, a segmented scan involves calling `SetSegment` followed by one of the combine functions, but the call to `SetSegment` is often left out, since segments are typically used more than once. Finally, segments are determined at the time the scan starts (on this node); changing the segments after that has no effect.

There are three kinds of segment boundaries; they form `SegmentType`:

```
typedef enum {
    NoBoundary, ElementBoundary, ArrayBoundary
} SegmentType;
```

The difference between `ElementBoundary` and `ArrayBoundary` requires an example. Assume that there are 16 processors (0 on the left), the operation is `ScanAdd`, and that segment boundaries are marked as `N`, `E`, or `A`:

```
Case 1:
    Value:    1 1 1 1   2 2 2 2   3 3 3 3   4 4 4 4
    Segment:  E N N N   E N N N   E N N N   E N N N

    Output:   0 1 2 3   0 2 4 6   0 3 6 9   0 4 8 12

Case 2:
    Value:    1 1 1 1   2 2 2 2   3 3 3 3   4 4 4 4
    Segment:  A N N N   A N N N   A N N N   A N N N

    Output:   0 1 2 3   4 2 4 6   8 3 6 9  12 4 8 12
```

Case 1 outputs the expected answer; the first node in a segment receives the identity value. However, case 2 is also useful (and matches the hardware). In case 2, the first node of a segment receives the value of the previous segment, which in this example is the sum of the previous segment. Thus, `ArrayBoundary` essentially means "give me the value from the previous processor, but don't forward that value to the next processor". This semantics is required when performing a scan on a "vector" that is wider than the partition and thus uses multiple elements per node. For example, a scan of a 64-element wide vector on 16 processors uses 4 elements per node. If the intended boundary is between the 2^{nd} and 3^{rd} elements, then the first two elements depend on the value from the previous node, and only the third element gets the identity value. The test code, `test-all.c`, uses both forms of segments.

`ATOMIC void SetSegment(SegmentType boundary)`

Sets the segment status for this node to `boundary`. If all nodes execute `SetSegment(NoBoundary)`, then the nodes form one large segment; this is the initial segment setting. It is fine for only a subset of the nodes to call `SetSegment`. Note that reductions (both regular and composite) ignore segment boundaries.

`ATOMIC SegmentType CurrentSegment(void)`

Returns the current segment setting for this node.

4.3.2 Combine Procedures

`int CombineInt(int value, CombineOp kind)`

Performs a global combine operation using variant `kind`; each PE contributes `value`. The return value is the result of the combine operation. This implicitly uses the current segment settings.

`ATOMIC void StartCombineInt(int value, CombineOp kind)`

Starts a split-phase combine operation. This implicitly uses the current segment settings.

`ATOMIC Bool QueryCombineInt(void)`

Returns true if and only if the last combine operation has completed.

`int CompleteCombineInt(void)`

Returns the result of the pending split-phase combine operation, possibly blocking.

```
void CombineVector(int to[], int from[],
    CombineOp kind, int num_elements)
```

Performs a vector combine operation. The input vector is `from` and the result is placed into `to`, which may be the same as `from`. This implicitly uses the current segment settings. (The order of `from` and `to` matches that of `memcpy` and its variants.)

`void StartCombineVector(int to[], int from[], CombineOp kind, int num_elements)`

Starts a split-phase vector combine operation. This implicitly uses the current segment settings.

`void CompleteCombineVector(void)`

Blocks until the pending split-phase vector combine operation completes. There is currently no `QueryCombineVector`.

4.4 Broadcast Operations

`ATOMIC void Broadcast(Word value)`

Broadcasts one word to all nodes.

`ATOMIC Bool QueryBroadcast(void)`

Returns true if and only if the pending broadcast has completed. This works with `BroadcastDouble` as well.

`Word ReceiveBroadcast(void)`

Returns the broadcast word. Should be called on all nodes except for the one that performed the broadcast.

`ATOMIC void BroadcastDouble(double value)`

Broadcast a double to all nodes.

`double ReceiveBroadcastDouble(void)`

Return the broadcast double. Should be called on all nodes except for the one that performed the broadcast.

`void BroadcastVector(Word to[], Word from[], int num_elements)`

Broadcast a vector of words. Fills in `to` as the words are received. Returns when all words are sent; not all may have been received.

`void ReceiveBroadcastVector(Word to[], int num_elements)`

Receive a vector of words into array `to`. Should be called on all nodes except for the one that performed the broadcast. For the broadcaster, the `to` array should be same as the call to `BroadcastVector`. Returns when all of the words have been received.

5 Composite Reductions

There are many useful operations that can be built on top of the CM-5's control-network hardware. Currently, Strata provides five such operations for three basic types: integers, unsigned integers, and single-precision floating-point numbers. These operations form the enumerated type `CompositeOp`:

```
typedef enum {
    Min, Max, Average, Variance, Median
} CompositeOp;
```

As with CMMD and the Strata control-network primitives, these reductions ignore the segment boundaries: all nodes form one large segment.[2] `Min`, `Max`, and `Average` are straightforward reductions. `Median` computes the median of the values contributed by the nodes. If there are an even number of values, then it returns the smaller of the two middle values.

`Variance` computes the sample variance of the values:

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}) = \frac{1}{n-1}\left[\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}\right]$$

where n is the size of the partition, $\bar{x}$ is the average, and x_i is the value contributed by the i^{th} node. The latter form is the one actually used. If the partition size is one, the reduction returns zero. The sample standard deviation is square root of the returned value.

There are three composite reduction procedures, one for each basic type:

`int CompositeInt(int value, CompositeOp op)`

This procedure applies the composite operation to integer values. The `Average` operation has the same overflow properties as a `ReduceAdd` of the same values. `Variance` is limited to the same precision as `float`.

`unsigned CompositeUint(unsigned value, CompositeOp op)`

[2] Future versions of Strata may provide segmented reductions; let us know if it is important to you.

This procedure applies the composite operation to unsigned integer values. The `Average` operation has the same overflow properties as a `ReduceUadd` of the same values. `Variance` is limited to the same precision as `float`.

`float CompositeFloat(float value, CompositeOp op)`

This procedure applies the composite operation to single-precision floating point values. It uses the CMMD procedure `CMMD_reduce_float` to ensure IEEE floating-point compatibility, and is provided primarily for completeness.

6 Data-Network Primitives

On the CM-5, an active message is a single-packet message in which the first word of the message specifies a procedure on the receiving processor. This procedure is called a *handler* and the handler's job is to remove the message from the network and incorporate it into the ongoing computation[vCGS92]. A CM-5 active-message packet consists of five words.

handler	arg1	arg2	arg3	arg4

When the receiving node polls the network and discovers an active message, the polling procedure invokes the `handler` with the four arguments `arg1`, `arg2`, `arg3`, and `arg4` as parameters.

6.1 Sending Active Messages

The CM-5 data network actually consists of two separate networks: the *right* network and the *left* network. Strata provides procedures to send and receive active messages on either network.

`void SendBothRLPollBoth(int proc, void (*handler)(), ...)`

Send an active message to processor `proc` using either network, and polling both networks. The active message is formed with `handler` as the first word and the remaining (up to) four parameters as `arg1`, `arg2`, `arg3`, and `arg4`. This procedure starts by trying to send the message on the right network and polling the right network. If the message does not get out, then it tries to send on the left network and polls the left network. This procedure continues this cycle until the message gets sent.

`void SendBothLRPollBoth(int proc, void (*handler)(), ...)`

This procedure starts by trying to send its message on the left network and polling the left network. If unsuccessful, it tries to send on the right network and polls the right network. This procedure repeats this cycle until the message gets sent.

void SendBothRLPollRight(int proc, void (*handler)(), ...)

This procedure sends an active message using either network, but it only polls the right network. It starts by trying to send its message on the right network and polling the right network. If the message doesn't get sent, then it tries to send on the left network (but doesn't poll the left network). This procedure repeats this cycle until the message gets sent.

void SendBothLRPollRight(int proc, void (*handler)(), ...)

This procedure starts by trying to send its message on the left network (but it doesn't poll the left network). If unsuccessful, it tries to send on the right network and polls the right network. This procedure repeats this cycle until the message gets sent.

void SendLeftPollBoth(int proc, void (*handler)(), ...)

This procedure sends an active message on the left network and polls both networks. It starts by trying to send its message on the left network and polling the left network. If the message does not get sent, then it polls the right network. This procedure repeats this cycle until the message gets sent.

Roughly, these procedures have the following CMMD 3.0 equivalents.

CMMD 3.0	**Strata**
`CMAML_request`	`SendLeftPollBoth`
`CMAML_reply`	`SendBothRLPollRight`
`CMAML_rpc`	`SendBothRLPollBoth`

(Actually `CMAML_reply` will only send its message on the right network.) When using any of the `SendBoth` variants, the choice between `SendBothRL` and `SendBothLR` can be made arbitrarily, but if these procedures are being called many times (especially in a tight loop), then higher performance is achieved by alternating between them.

6.2 Receiving messages

Messages are removed from the network and appropriate handlers are invoked by the following polling procedures.

void PollLeft(void)

This procedure checks the left network, and if there is a pending message, it removes the message and calls the appropriate handler.

void PollRight(void)

This procedure checks the right network, and if there is a pending message, it removes the message and calls the appropriate handler.

`void PollBoth(void)`

This procedure checks both networks and pulls out at most one message from each. It is equivalent to `PollLeft` followed by `PollRight`.

`void PollLeftTilEmpty(void)`

This procedure keeps removing messages from the left network and invoking the appropriate handlers until it finds that no further messages are pending.

`void PollRightTilEmpty(void)`

This procedure keeps removing messages from the right network and invoking the appropriate handlers until it finds that no further messages are pending.

`void PollBothTilEmpty(void)`

This procedure keeps removing messages from both networks and invoking the appropriate handlers until it finds that no further messages are pending on either network.

`void PollLeftThenBothTilEmpty(void)`

This procedure begins by polling the left network. If it finds no message pending, then it returns. If it finds a pending message, then it continues polling both networks until it finds no further messages are pending on either network.

`void PollRightThenBothTilEmpty(void)`

This procedure begins by polling the right network. If it finds no message pending, then it returns. If it finds a pending message, then it continues polling both networks until it finds no further messages are pending on either network.

In general, frequent polling with `PollBothTilEmpty` is recommended. In a loop, however, higher-performance is achieved by alternating between `PollLeftThenBothTilEmpty` and `PollRightThenBothTilEmpty`.

7 Block Transfers

Block transfers are performed by sending special messages called *Xfer messages* special data structures called *ports* at the receiving processor. A port is a structure defined in Strata as follows.

```
typedef struct {
    Word  *base;
    int    count;
    void (*handler)();
    int    user1;
    int    user2;
    int    user3;
    int    user4;
    int    user5;
} StrataPort;
```

Strata provides each processor with a global array of `STRATA_NUM_PORTS` ports (currently 4096). This array is called the `StrataPortTable` and is defined as

```
StrataPort StrataPortTable[STRATA_NUM_PORTS];
```

Ports are designated by number, that is, by index into the `StrataPortTable`. To receive a block transfer at a port, the port's `base` field must be initialized to point to a block of memory into which the block transfer data can be stored. Also, the port's `count` field must be set to the number of words to be received in the block transfer. As the block transfer data arrives, the count value gets decremented, and when the count reaches zero, the procedure specified by the `handler` field (if not `NULL`) gets invoked with the port number as its single parameter. The other five fields in the port structure, `user1` through `user5`, are available for arbitrary use.

Block transfer data is received by the same polling procedures that receive active messages as described in the previous section.

The following two procedures are used to send a block transfer.

`void SendBlockXferPollBoth(int proc, unsigned port_num, int offset, Word *buffer, int size)`

Sends `size` words starting at `buffer` to the port `port_num` at the destination processor `proc` and stores the data in the destination processor's memory starting at the address given by adding `offset` (in words) to the port's base value, that is, at address (`StrataPortTable[port_num].base + offset`). The data is sent on both networks, and this procedure will poll both networks.

`void SendBlockXferPollRight(int proc, unsigned port_num, int offset, Word *buffer, int size)`

Sends `size` words starting at `buffer` to the port `port_num` at the destination processor `proc` and stores the data in the destination processor's memory starting at the address given by adding `offset` (in words) to the port's base value, that is, at address (`StrataPortTable[port_num].base + offset`). The data is sent on both networks, but this procedure only polls the right network.

The port number, `port_num`, must be between 0 and `STRATA_NUM_PORTS - 1`. The `offset` is a signed value and must be at least `STRATA_MIN_XFER_OFFSET` (currently -2^{19}), and the sum `offset + size` must be no larger than `STRATA_MAX_XFER_OFFSET` (currently $2^{19}-1$). Thus, a block transfer can consist of up to 2^{20} words (with the port's `base` field pointing to the middle of the destination block). (To send a larger block, use more than one port.)

Notice that all quantities are in terms of words not bytes.

The block transfer is most efficient when the source block address (given by `buffer`) and the destination block address (given by adding `offset` to the port's `base` value) have the same alignment. That is, when both are `DoubleWord` aligned or both are not `DoubleWord` aligned (but are, of course, `Word` aligned).

Before sending a block transfer, the receiver must have set the `base` field of the target port, but the `count` and `handler` fields can be set by the sending processor before, after, or during the block transfer. The sending processor sets these fields by sending an ordinary active message (as described in the previous section) that invokes a handler on the target processor to set these fields. Strata provides just such a handler.

`HANDLER void StrataXferHeaderHandler(int port_num, int size, void (*handler)())`

This handler increments the `count` field of port `port_num` by `size`, and if the port's `handler` field is `NULL`, sets the `handler` field to `handler`. It then checks to see if the `count` field is zero, and if so, it invokes the procedure given by the `handler` field (if not `NULL`) with `port_num` as its single parameter.

If the sending processor is going to set the port's `handler` field, then the destination processor should initialize the port's `handler` field to `NULL`. And if the sending processor is going to set the port's `count` field, then the destination processor should initialize the port's `count` field with zero. The sending processor then, in addition to calling `SendBlockXferPollBoth` or `SendBlockXferPollRight` to send the actual data block, must send an active message that invokes `StrataXferHeaderHandler`. Some of the block data may arrive at the destination processor before this active message, and in this case, the `count` field will actually go negative (it is initialized at zero). Then when the active message arrives, the `count` field gets incremented — hopefully to some non-negative value. Of course, the active message may arrive even after *all* of the block transfer data; in this case, incrementing the `count` field should bring it to zero, and for this reason, `StrataXferHeaderHandler` checks to see if the `count` field is zero and takes appropriate action.

As a typical example, a receiving processor might set the `base` field of a port to point to a (suitably large) buffer, `dest_buffer`, and initialize the port's `count` field to 0 and its `handler` field to `NULL`. Then the sending processor can send `size` words from its `source_buffer` to the receiving processor's `dest_buffer` (assuming the sending processor knows that the port is number `port_num`) and cause the procedure `handler` to be invoked on the receiver when the transfer is complete with the following pair of procedures.

```
SendBothRLPollBoth(dest_proc, StrataXferHeaderHandler, port_num,
                   size, handler);
SendBlockXferPollBoth(dest_proc, port_num, 0, source_buffer, size);
```

7.1 For the curious

The procedures `SendBlockXferPollBoth` and `SendBlockXferPollRight` send their data through a sequence of single-packet Xfer messages. An Xfer message manages to pack 4 words of payload data into the 5 word CM-5 data-network packet.

port_and_offset	data1	data2	data3	data4

The port number and offset values are packed into a single word `port_and_offset`. When such an Xfer message arrives at its destination processor (and the destination processor receives it by polling), a special Xfer handler routine is invoked. This routine splits the `port_and_offset` into its two components: `port_num` and `offset`. Then it stores the four data words, `data1` through `data4`, at consecutive addresses starting at the address given by adding `offset` (in words) to the port's base address, that is, at address `StrataPortTable[port_num].base + offset`. It then subtracts 4 from the port's `count` field, and takes appropriate action if the `count` is zero.

Port number and offset values are packed into and unpacked from a single word with the following functions.

`ATOMIC unsigned PortAndOffsetCons(unsigned port_num, int offset)`

Packs `port_num` and `offset` into a single word. The `port_num` must be between 0 and `STRATA_NUM_PORTS - 1`, and the `offset` must be between `STRATA_MIN_XFER_OFFSET` and `STRATA_MAX_XFER_OFFSET`. (Currently, `port_num` lives in the low 12 bits and `offset` lives in the high 20 bits of `port_and_offset`, so `STRATA_NUM_PORTS` equals 4096, `STRATA_MIN_XFER_OFFSET` equals -2^{19}, and `STRATA_MAX_XFER_OFFSET` equals $2^{19} - 1$.

`ATOMIC unsigned PortAndOffsetPort(unsigned port_and_offset)`

Extracts the `port_num` from `port_and_offset`.

`ATOMIC int PortAndOffsetOffset(unsigned port_and_offset)`

Extracts the `offset` from `port_and_offset`.

Single-packet Xfer messages are sent with the following procedures.

`void SendXferBothRLPollBoth(int proc, unsigned port_and_offset, Word data1, Word data2, Word data3, Word data4)`

```
void SendXferBothLRPollBoth(int proc, unsigned port_and_offset,
    Word data1, Word data2, Word data3, Word data4)
```

```
void SendXferBothRLPollRight(int proc, unsigned port_and_offset,

    Word data1, Word data2, Word data3, Word data4)
```

```
void SendXferBothLRPollRight(int proc, unsigned port_and_offset,

    Word data1, Word data2, Word data3, Word data4)
```

Each single-packet Xfer message carries exactly four words of payload. For a block transfer with a number of words that is not a multiple of four, extra messages must be sent. These extra messages can be sent with ordinary active messages since they only need 1, 2, or 3 words of payload. Strata provides handlers to deal with these active messages.

```
HANDLER StrataXferPut3Handler(unsigned port_and_offset,
    Word data1, Word data2, Word data3)
```

```
HANDLER StrataXferPut2Handler(unsigned port_and_offset,
    Word data1, Word data2)
```

```
HANDLER StrataXferPut1Handler(unsigned port_and_offset,
    Word data1)
```

These handlers store the data words into the appropriate location, subtract the appropriate value from the port's `count` field, and take appropriate action if the `count` is zero.

None of these functions, procedures, or handlers are needed if you use `SendBlockXfer-PollBoth` or `SendBlockXferPollRight`. These two procedures do it all for you.

8 Multi-Block Transfer

Strata provides an asynchronous block-transfer protocol to support multiple block transfers. This interface allows Strata to interleave packets from several different block transfers, which improves the efficiency of the network and increases the net effective bandwidth.

Pending transfers are identified with handles of type `ABXid`.

`ABXid AsyncBlockXfer(int proc, unsigned port,`
`int offset, Word *buffer, int size,`
`void (*complete)(ABXid id, Word *buffer))`

Initiate an asynchronous block transfer to port `port` of processor `proc`. The argument correspond to normal block-transfer routines, except for the additional argument `complete`. This function, if non-`NULL`, is called upon completion of the *sending* of this transfer. The return value is an identifier that is used by the following routines.

`void ServiceAllTransfers(int rounds)`

This procedure services all of the pending asynchronous block transfers, sending 2*`rounds` packets for each transfer. If `rounds` is -1, then this procedures synchronously completes all pending transfers. The preferred use of this routine is to set up all of the transfers (or a reasonable size subset), and then to call `ServiceAllTransfers(-1)` to actually send them. For target distributions that prevent explicit scheduling of transfers, this technique provides about doubles the performance of just sending the transfers synchronously.

`void CompleteTransfer(ABXid id)`

This synchronously completes the named transfer.

`int GetABXproc(ABXid id)`
`unsigned GetABXport(ABXid id)`
`int GetABXoffset(ABXid id)`
`Word *GetABXbuffer(ABXid id)`
`int GetABXremaining(ABXid id)`

These routines access the corresponding information regarding the named transfer.

9 Debugging Operations

9.1 Printing in Handlers

`ATOMIC int Qprintf(const char *format, ...)`

Queue up a printf to `stdout`. This can be used anywhere, including within handlers, because it does not poll and does not allocate memory. The return value is the number of characters in the message. `Qprintf` allows you to insert debugging statements *without* changing the atomicity of the surrounding code.

`ATOMIC int Qfprintf(FILE *out, const char *format, ...)`

Queue up an fprintf to file `out`. Otherwise identical to `Qprintf`.

`int EmptyPrintQ(void)`

Outputs the queued messages. This does poll and thus should not be called from within a handler. The return value is the number of messages that had been queued up.

A Note On `pndbx`

When using `pndbx` to debug a Strata application, it is often useful to read the queued-up print calls. Strata provides two global variables to help with this. First, the variable `char *StrataQueue` points to the buffer containing the queued-up text. The strings are null-terminated and packed continuously. Executing print StrataQueue from `pndbx` prints the first string (only). The variable `int StrataQueueLength` contains the number of queued-up strings. To see all of the strings, first use print &StrataQueue[0] to get an address, say 0x1cb4a0, then use 0x1cb4a0/256c to dump the buffer as list of 256 characters (the 256 is arbitrary — large queues may require a larger number). Finally, if the program is active, it is often safe to execute call EmptyPrintQ() to output the strings; this is less useful if the output is going to a file rather than the terminal.

9.2 Assertion Checking

Macro `assert(x)`

Calls `StrataFail` with a nice error message if `x` evaluates to 0. This replaces the normal `assert` macro both to improve the message and to ensure a clean exit.

9.3 Debugging Mode and Logging

Strata provides a debugging mode that performs safety checks and atomic logging. To use debugging mode, you must compile with the `-DSTRATA_DEBUG` flag and link with `-lstrataD` instead of `-lstrata`. Debugging mode uses several global variables:

`Bool StrataLogging`	True iff linked with the debugging library.
`Bool LogBarrier`	Log barrier info, default is `False`
`Bool LogBroadcast`	Log broadcast info, default is `False`
`Bool LogCombine`	Log combine info, default is `False`
`Bool LogQprintf`	Log Qprintf info, default is `False`

The log appears in file `CMTSD_printf.pn.`*number*, where *number* is the process id of the host. Strata programs print out the exact name on exit (unless killed). The log is always up to date; for example, when a program hangs, the last log entry for each node often reveals the nature of the problem. Strata comes with a perl script called `loglast` that outputs the last entry for each node.

Users may log their own messages in addition to those provided by the debugging library:

`ATOMIC void StrataLog(const char *fmt, ...)`

The printf-style message is added to the log. Note that all log entries are prepended with the processor number, so there is no need explicitly print it. `StrataLog` may be used anywhere, even when not using the debugging library. It should be viewed as an expensive function.

9.4 Timing Functions

`ATOMIC INLINE unsigned CycleCount(void)`

Returns the value of the cycle counter.

`ATOMIC INLINE unsigned ElapsedCycles(unsigned start_count)`

Returns the elapsed time in cycles given the starting time. The intended use is:

```
start = CycleCount();
...
elapsed = ElapsedCycles(start);
```

The routines are calibrated so that if used in an application compiled with `-O`, the overhead of the two calls is exactly subtracted out. Thus if the two calls are adjacent, the elapsed time is zero.[3] These routines will break if the code is time sliced between the calls, but this is extremely unlikely for (the intended) short timings. For example, a 10,000 cycle event has roughly a 1 in 330 chance of being time sliced given the usual time-slice interval of 0.1 seconds (and zero chance if the machine is in dedicated mode).

`ATOMIC INLINE double CyclesToSeconds(unsigned cycles)`

Converts a time in cycles into seconds (not microseconds).

`DoubleWord CurrentCycle64(void)`

[3] With `-O`, the null timing case compiles to two consecutive reads of the cycle counter; the subtraction assumes this case and the result is almost always zero. The exception is when the two reads are in different cache lines and the cache line of the second read is not loaded. In this case, the timing includes the cache miss and typically returns 28 or 29 cycles. The point of all this is that you should be aware of cache behavior for short timings.

Returns the number of cycles elapsed since initialization (via `StrataInit`). This is not as accurate as `ElapsedCycles`, but it works across time slices (modulo some OS bugs) and provides 64-bit resolution. The implementation uses the same underlying code as the CMMD timers and is thus exactly as accurate.

`ATOMIC INLINE unsigned CurrentCycle(void)`

Same as `CurrentCycle64` except that it returns the elapsed cycles since the initialization or the last call to `ResetCurrentCycle`. Like `ElapsedCycles` this procedure includes cycles due to time slicing; it is thus primarily useful in dedicated mode.

`void ResetCurrentCycle(void)`

This sets the `CurrentCycle` base time to zero. It is a synchronous operation and must be called on all nodes. The primary use of this routine is to get 32 bits of useful times in the middle of a program that runs for more the 2^32 cycles. It affects the timestamps used by the graphics module, which allows users to get a 2^32-cycle state-graph of their long-running program for any single such continuous interval.

10 Graphics

Version 2.0 supports a subset of the PROTEUS graphics capabilities. The primary form of graph currently supported by Strata is the *state graph*, which plots the state of each processor (as a color) versus time (in cycles). There are several limitations on state graphs: 1) there can be at most 16 states, 2) the program can run for at most 2^32 cycles after the call to `InitTrace`, and 3) there is a limit on the number of state changes per processor (although there are workarounds for this one).

The graphics module creates a trace file that is read by the program `stats`. The interpretation of the file is determined by a *graph-file specification* that specifies the names of states and how to build each graph (normally there is only one graph for Strata).

The easiest way to understand Strata's graphics is to play with the example radix-sort program (`radix.c`). It generates a trace file called `radix.sim` that can be examined with:

```
stats -f radix.sim -spec cmgraph
```

The second pair of arguments identifies the graph-specification file. Figure 2 shows the state graph for radix sort. A seperate document covers the use of `stats` and the graph-specification language; `stats -help` provides some information as well.

State 0 always means "idle" and state 1 always means "busy", although the real meaning of these states is up to the application. In debugging mode, Strata uses states 2 through 7 as follows:

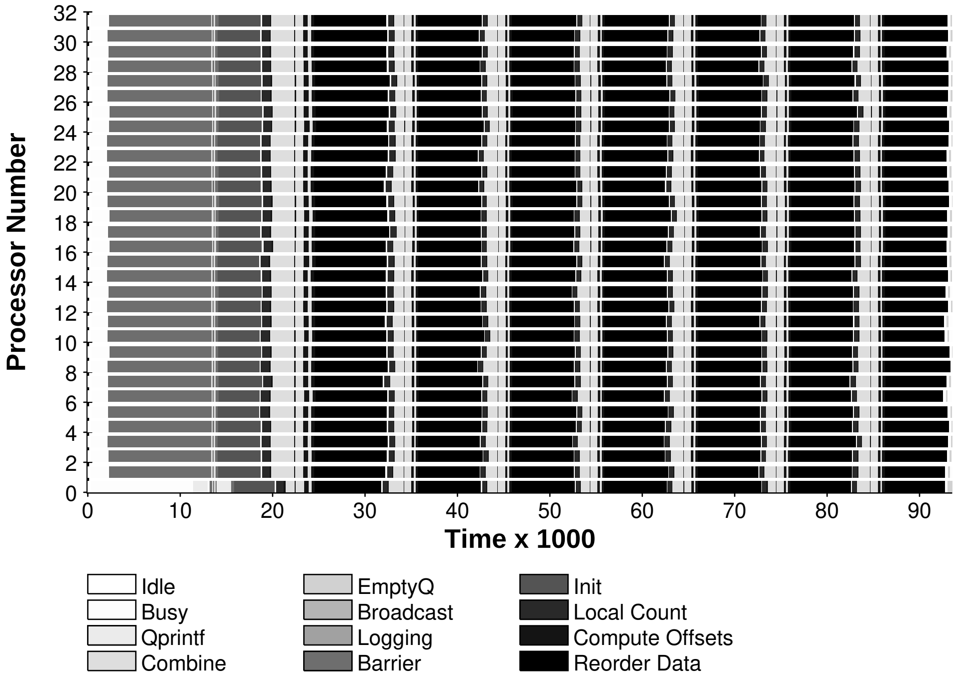

Figure 2: The state graph for radix sort.

State	Meaning
2	Logging
3	Qprintf
4	EmptyPrintQ
5	Broadcasting
6	Combine Operation
7	Barrier

These states can be redefined via `strata.h` and may even be combined if the user needs more application-specific states.

Generating the trace file involves only three procedures:

`void InitTrace(const char *file, const char *title)`

> This initializes the graphics module; it must be called before `StrataInit` and it must be called on all processors. It is a global operation. The first argument is the name of trace file. The second argument is the title of the simulation; if non-`NULL`, this title will appear as a subtitle on the graphs (the primary title is determined by the graph-specification file).

`ATOMIC int SetState(int state)`

> This changes the state of this processor to `state` and returns the old state. The state change is timestamped with the current cycle time.

`Bool RecordStates(Bool on)`

Turns state recording (via `SetState`) on or off (for the local node only). This is primarily useful for turning off state generation after the interesting part of the program completes. For example, to record states for one section in the middle of a long-running program, use `ResetCurrentCycle()` to mark the beginning of the section, and `RecordStates(False)` to mark the end.

`void OutputStateLog(void)`

Normally, the state changes are kept in local memory until the program exits, at which point they are moved to disk. Currently, the buffering allows at most 2048 state changes. `OutputStateLog` empties the buffer so that more events can be generated. This is a synchronous operation and must be executed on all nodes; it should be viewed as a very expensive barrier. Most programs do not need to call this procedure.

11 Acknowledgments

Thanks to Mike Halbherr, Chris Joerg, Ulana Legedza, Tom Leighton, Charles Leiserson, Arthur Lent, Frans Kaashoek, Bradley Kuszmaul, Dave Park, Ryan Rifkin, Carl Waldspurger, Bill Weihl, and Yuli Zhou.

References

[PM88] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10), October 1988.

[TMC93] Thinking Machines Corporation. *CMMD Reference Manual, Version 3.0*, May 1993.

[vCGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the* 19^{th} *International Symposium on Computer Architecture (ISCA '92)*, pages 256–266, May 1992.

A Prototype Summary

Page	Basics
6	`void StrataInit(void)`
7	`NORETURN StrataExit(int code)`
7	`NORETURN StrataFail(const char *fmt, ...)`
7	`ATOMIC void ClearMemory(void *region, unsigned length_in_bytes)`
7	`ATOMIC unsigned Random(void)`
7	`ATOMIC unsigned SetRandomSeed(unsigned seed)`

	Timing
22	`ATOMIC unsigned CycleCount(void)`
22	`ATOMIC unsigned ElapsedCycles(unsigned start_count)`
22	`ATOMIC double CyclesToSeconds(unsigned cycles)`
22	`DoubleWord CurrentCycle64(void)`
23	`ATOMIC unsigned CurrentCycle(void)`
23	`void ResetCurrentCycle(void)`

	Global OR
8	`Bool GlobalOR(Bool not_done)`
8	`ATOMIC void StartGlobalOR(Bool not_done)`
8	`ATOMIC Bool QueryGlobalOR(void)`
8	`Bool CompleteGlobalOR(void)`
8	`ATOMIC void SetAsyncGlobalOR(Bool not_done)`
8	`ATOMIC Bool GetAsyncGlobalOR(void)`

	Barriers
8	`void Barrier(void)`
8	`ATOMIC void StartBarrier(void)`
8	`ATOMIC Bool QueryBarrier(void)`
8	`void CompleteBarrier(void)`

	Combine Operations
10	`ATOMIC void SetSegment(SegmentType boundary)`
10	`ATOMIC SegmentType CurrentSegment(void)`
10	`int CombineInt(int value, CombineOp kind)`
10	`ATOMIC void StartCombineInt(int value, CombineOp kind)`
10	`ATOMIC Bool QueryCombineInt(void)`
10	`int CompleteCombineInt(void)`
10	`void CombineVector(int to[], int from[], CombineOp kind, int num_elements)`
11	`void StartCombineVector(int to[], int from[], CombineOp kind, int num_elements)`
11	`void CompleteCombineVector(void)`

```
12    int CompositeInt(int value, CompositeOp op)
12    unsigned CompositeUint(unsigned value, CompositeOp op)
13    float CompositeFloat(float value, CompositeOp op)
```

Broadcast Operations

```
11    ATOMIC void Broadcast(Word value)
11    ATOMIC Bool QueryBroadcast(void)
11    Word ReceiveBroadcast(void)
11    ATOMIC void BroadcastDouble(double value)
11    double ReceiveBroadcastDouble(void)

11    void BroadcastVector(Word to[], Word from[], int num_elements)
12    void ReceiveBroadcastVector(Word to[], int num_elements)
```

Sending Active Messages

```
13    void SendBothRLPollBoth(int proc, void (*handler)(), ...)
13    void SendBothLRPollBoth(int proc, void (*handler)(), ...)
14    void SendBothRLPollRight(int proc, void (*handler)(), ...)
14    void SendBothLRPollRight(int proc, void (*handler)(), ...)
14    void SendLeftPollBoth(int proc, void (*handler)(), ...)
```

Polling

```
14    void PollLeft(void)
14    void PollRight(void)
15    void PollBoth(void)
15    void PollLeftTilEmpty(void)
15    void PollRightTilEmpty(void)
15    void PollBothTilEmpty(void)
15    void PollLeftThenBothTilEmpty(void)
15    void PollRightThenBothTilEmpty(void)
```

Block Transfers

```
16    void SendBlockXferPollBoth(int proc, unsigned port_num,
                int offset, Word *buffer, int size)
16    void SendBlockXferPollRight(int proc, unsigned port_num,
                int offset, Word *buffer, int size)

18    ATOMIC unsigned PortAndOffsetCons(unsigned port, int offset)
18    ATOMIC unsigned PortAndOffsetPort(unsigned port_and_offset)
18    ATOMIC int PortAndOffsetOffset(unsigned port_and_offset)

18    void SendXferBothRLPollBoth(int proc, unsigned port_and_offset,
                Word data1, Word data2, Word data3 Word data4)
19    void SendXferBothLRPollBoth(int proc, unsigned port_and_offset,
                Word data1, Word data2, Word data3 Word data4)
19    void SendXferBothRLPollRight(int proc, unsigned port_and_offset,
```

```
             Word data1, Word data2, Word data3 Word data4)
19   void SendXferBothLRPollRight(int proc, unsigned port_and_offset,
             Word data1, Word data2, Word data3 Word data4)

17   HANDLER void StrataXferHeaderHandler(int port_num, int size,
             void (*handler)())
19   HANDLER void StrataXferPut3Handler(unsigned port_and_offset,
             Word data1, Word data2, Word data3)
19   HANDLER void StrataXferPut2Handler(unsigned port_and_offset,
             Word data1, Word data2)
19   HANDLER void StrataXferPut1Handler(unsigned port_and_offset, Word data1)
```

Multi-Block Transfers

```
20   ABXid AsyncBlockXfer(int proc, unsigned port, int offset, Word *buffer,
             int size, void (*complete)(ABXid id, Word *buffer))
20   void ServiceAllTransfers(int rounds)
20   void CompleteTransfer(ABXid id)

20   int GetABXproc(ABXid id)
20   unsigned GetABXport(ABXid id)
20   int GetABXoffset(ABXid id)
20   Word *GetABXbuffer(ABXid id)
20   int GetABXremaining(ABXid id)
```

Debugging

```
20   ATOMIC int Qprintf(const char *format, ...)
21   ATOMIC int Qfprintf(FILE *out, const char *format, ...)
21   int EmptyPrintQ(void)
21   Macro assert(Bool x)
22   ATOMIC void StrataLog(const char *format, ...)
```

Graphics

```
24   void InitTrace(void)
24   ATOMIC int SetState(int state)
25   Bool RecordStates(Bool on)
25   void OutputStateLog(void)
```