The Design of a Memory Management Subsystem for the FFP Machine

TR 89-017

Raj K. Singh
Vernon L. Chi

The FFPM Project and the
Microelectronic Systems Laboratory
The University of North Carolina
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

This work was sponsored by NSF/CISE/MIPS under grant number MIP-8702277 and ONR under contract number N00014-86-K-0680. The authors gratefully acknowledge the valuable contributions of the entire FFPM project team: Edoardo Biagioni, Tai-sook Han, Gyula Magó, William Partain, David Plaisted, Jan Prins, Bruce Smith, and Donald Stanat of UNC, and Ting Feng, Charles Molnar, and Fred Rosenberger of Washington University.

Reference and title

TR-89017
The Design of a Memory Management Subsystem for the FFP Machine

Publication history

Version 1.0: 1 May 1989

Design of a Memory Management Subsystem for the FFP Machine

Raj K. Singh and Vernon L. Chi

v1.0: 1 May 1989

1	Introduction	1
2	System Components and Nomenclature	2
3	Parameter binding strategies	3
4	Data storage strategies	4
4.1	S-cell size	5
4.2	Buffer design	6
5	Data movement strategies	9
6	Cloning strategies	10
6.1	Cloning time	10
6.2	Cloning direction	11
7	The transfer protocol	11
7.1	Pure shifting protocol	12
7.2	Cloning protocol	12
7.3	Algorithmic description of the protocol	12
8	Hardware design	17
9	Summary	18
10	Future directions	18
11	Appendix A: Lazy data movement storage preparation algorithm	19
12	Appendix B: Modified flow numbers	20
13	References	21

Design of a Memory Management Subsystem for the FFP Machine

Raj K. Singh and Vernon L. Chi

v1.0: 1 May 1989

1 Introduction

The organization and management of data storage in a computer system are important factors in the design of both the machine and its operating system. Computer memory systems have historically evolved from simple on-line storage devices to complex hierarchies of media having a wide price/performance range. These hierarchies invariably attempt to exploit the property of data locality in program execution to enable the performance of the entire memory system to approach that of its fastest (and most expensive) part.

In the case of uniprocessors, we see very expensive, fast caches backed up by slower main memory. This in turn is backed up by disk files which are backed in turn by tapes. Storage organization is balanced under various policies which typically address issues such as partitioning (rigid or dynamic), multiprogramming, and mapping of the user program onto the memory. Various intricate storage management mechanisms have been developed to implement storage organization strategies by efficiently moving data within and between levels of the hierarchy with the objective of providing the highest possible performance at the lowest possible cost. Examples are virtual memory and paging mechanisms.

The same economic forces motivate development of memory systems for multiprocessors. But the problem is exacerbated by the coupled concern of moving data between equivalent data stores at the same hierarchical level to achieve the needed processor to memory bandwidth

This report describes the design of a high performance memory management I/O subsystem for a non-shared memory MIMD architecture called the FFP-Machine (FFPM) [1]. It is optimized for inter processor communications, but also may provide I/O for an ensemble of processing elements.

The FFPM is a fine-grained MIMD machine organization which may be implemented using fine-grained hardware cells, or more economically emulated by coarser grained physical resources [2]. We address herein the issues of inter-cell communication for the fine-grained case. The discussion of additional complexities imposed by a coarse-grained implementation is outside the scope of this document.

The FFPM cycle consists of storage management, partitioning, and execution. This report focuses on the storage management phase. Storage management in the FFPM is composed of two sub-phases, storage preparation and data movement. During the preparation phase, computations are performed to set parameters used to control the data movement. A thorough treatment of storage preparation and a discussion of the algorithm used can be found in [3] and will not be treated here. In this report, we assume the availability of these pre-computed parameters, and concentrate on issues of data movement.

In Section 2 we give a brief structural overview of a processing element of the FFPM called the L-cell and introduce some basic nomenclature. Section 3 discusses how the data movement parameters computed during storage preparation are packaged for use during data movement. Design strategies for data storage, movement, and space acquisition are treated in Sections 4, 5, and 6, respectively. Section 7 describes a protocol for communication between L-cells. Finally, Section 8 presents an abstract hardware design for implementing the protocol.

2 System Components and Nomenclature

A high-level architectural block diagram of a fine-grained L-cell is shown in Fig. 1. The internal data bus is 8-bits wide, as are the lateral connections to neighboring L-cells. The CPU is an ordinary instruction set processor such as a Z80. The memory and the system bus are time shared between the cell's CPU and its I/O devices on a priority basis.

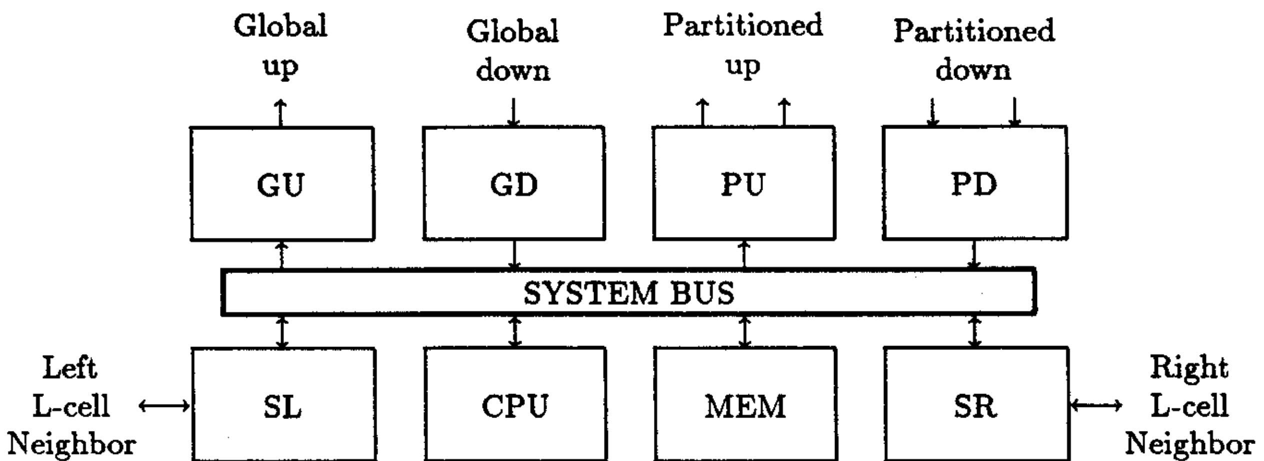

Figure 1: L-cell functional block diagram

The **GU**, **GD**, **PU**, and **PD** blocks represent communications interfaces used in the computation phases of the FFPM operation. They are not used during data movement and thus are of no concern to this discussion.

The Left-shifter (**SL**) and Right-shifter (**SR**) blocks represent bidirectional L-cell I/O ports, with dedicated controller devices for inter-cell communications. The SR port of each L-cell is connected to the SL port of its right hand neighbor such that the L-cells form a linearly connected array called the **L-array**.

These connections are low level synchronous, i.e., data transfers occur synchronously on a specific phase of the global system clock; but they are also high-level asynchronous, i.e., every transfer is locally controlled using full hand-shaking with a send-acknowledge protocol across each inter-cell connection.

Although the SL and SR ports in an L-cell communicate independently of each other, their operation is controlled by a finite-state machine to avoid memory over- and/or underruns. The operation of this control is critical to the data movement performance of the L-array during storage management.

The shifter devices (SL and SR) serve to shift Symbol-cells (S-cells) from one L-cell to another. An **S-cell** (sometimes called an **S-image**) is a data structure which represents the state of an FFP primitive symbol as stored in the L-array. During the execution phase, each L-cell contains exactly one S-cell in its **S-buffer** which is part of MEM in Fig. 1.

During data movement, there are five possible non-trivial combinations of operations for the two shifters of an L-cell:

1) shift current S-cell out while shifting new S-cell in
2) shift current S-cell out (without shifting anything in)
3) shift new S-cell in (without shifting anything out)
4) shift S-cell directly from input to output (without using S-buffer known as shunt through)
5) concurrently shift copies of current S-cell out both sides

Since the shifting devices are bidirectional, this allows nine distinct functional shifting modes: four each to the right and to the left, and one bidirectional. In any case, a device shifting out is called a **sender** and a device shifting in is called a **receiver**.

Some data may move to the right while other data may move to the left in the L-array during storage management. These data may be considered as being composed of strings of S-cells constituted such that each string moves, if at all, either entirely to the right or entirely to the left. These are called **right-shift-strings**, and **left-shift-strings**, respectively.

New instances of S-cells may be **cloned** during data movement. Each instance receives a unique clone number such that after data movement the strings of clones are sequentially numbered from left to right, with the leftmost being numbered zero.

During the data movement phase, a well defined number of S-cells will be shifted across any given inter-cell connection. This number is called the **flow number** for that connection (and for its associated SL and SR ports). A negative flow number represents left shifts, while a positive number represents right shifts. If the flow number is zero, no S-cells shift across the connection during data movement.

Blockage is a condition where local congestion temporarily prevents data transfers between L-cells during data movement. It is a major detractor from performance during storage management, and is the object of much of the discussion in this report.

The phenomenon of blockage can be compared with automotive traffic flow. Once the traffic is blocked, unblocking at the front does not enable all the traffic to begin simultaneously moving again; rather the resumed movement must bubble back through the queue of cars. This obviously has an adverse effect on performance, so we attempt to avoid blockage as much as possible.

In general blockage cannot be avoided altogether. Nevertheless, its effect on the cost of data movement can be minimized by a careful choice of protocol, shifting hardware, and buffering strategy.

3 Parameter binding strategies

During storage preparation, the full power of the FFPM is used to compute what the final destinations of the existing and new S-cells should be, given the demands of existing expressions under evaluation and expressions awaiting input to the machine. It is also used

to translate this destination information into parameters for controlling the movement of S-cells during the data movement phase.

It is desirable to maximize the speed of data movement due to the linear time nature of L-array communications (in contrast to the primarily logarithmic communication time consumed by computational cycles). Therefore, the data movement phase is implemented in special purpose shifter hardware without CPU intervention. An efficient design requires a simple algorithm to interpret these parameters, else the shifter cost rapidly gets out of hand.

A major decision is whether the parameters should be bound to S-cells or to L-cells (some hybrid of these bindings is also possible). This radically affects the shifter design. If they are bound to S-cells, the shifters must be adept at extracting, interpreting, manipulating, and maintaining them on the fly as S-cells flow into and out of the L-cells. If they are bound to the L-cells, registers containing them provide immediate and simple access to all of them at any time, even in parallel, if necessary. This is a compelling advantage in the design of simple, efficient hardware shifters.

Examples of S-cell-bound parameters are S-image data fields representing absolute destination addresses, relative displacement addresses, etc. Examples of L-cell-bound parameters are the flow numbers. Under some design strategies, a hybrid might require flow numbers and an S-image field representing a clone count, or at least a clone flag.

Since the complexity of hardware that can deal with S-cell-bound parameters tends to far exceed that of hardware to deal with L-cell-bound parameters, we are strongly influenced us in favor of the latter.

In this report, we take flow numbers to be the only parameters other than S-cell word-counts controlling data movement. They are set directly into registers in the L-cell by the L-cell's CPU during storage preparation, and maintained on the fly by the shifter hardware during data movement such that they have all become zero at the completion of data movement. Word-counts are invariant during dat movement, and are stored as the first field of each S-cell for easy and timely extraction by the shifters during data movement.

Figure 2 shows the flow numbers computed during storage preparation for an L-array of size 16. Here, the sub-string in cells 1 through 3 is an example of a left-shift-string, while the sub-strings in cells 9 through 12 and 13 through 16 are examples of right-shift-strings. The sub-string in cells 4 through 8 is a special case, consisting of both left- and right-shift-strings. Such a situation may only occur where cloning is in progress.

4 Data storage strategies

The purpose of data movement is to re-allocate L-cell resources to expressions that will require additional space during the next execution phase. This is accomplished by cloning enough copies of S-cells in expressions needing space, and moving them away from their original S-cells (which may themselves move). In the process, space occupied by empty S-cells is reclaimed.

Optimization of the data movement can be characterized as a 1-D minimax flow problem. The optimal solution is one that minimizes the time required for data movement.

Before data movement

L-cell number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
L-cell contents			A				B		C				D		E	

Flow numbers 0 −1 −2 0 −1 −2 −3 1 0 3 2 1 0 2 1 1 0

After data movement

L-cell number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
L-cell contents	A_0	A_1	A_2	B_0	B_1	B_2	B_3	B_4	C_0	C_1	C_2	C_3	D_0	D_1	D_2	E

Flow numbers 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 2: Example of flow numbers

For equal sized S-images, this means minimizing the maximum distance traveled by any S-image.

A straightforward approach would conceptually place the S-image in each L-cell in a bidirectional shift register which is connected to the SR and SL ports. Thus, data could be conditionally shifted, one character per clock cycle, until all S-cells in the L-array were properly relocated.

But S-cell data must be randomly accessible during the computation phases, and must therefore be stored in RAM. Hence, it is more economical to implement the shift register function in RAM. This incurs the usual overhead of bus contention and synchronization. Thus the shifting of each character of data from cell to cell may require several clock cycles.

Because of the relatively small size of the memory per L-cell and a design of the kernel software having a single-thread of control, the memory was segmented with a fixed mapping. Thus, the S-buffers have the same buffer location and size in all L-cells.

4.1 S-cell size

The actual amount of data needed to completely represent an S-cell varies with the nature of the symbol and execution progress of the microcode. During data movement, it is possible to move only the relevant data; however the necessary hardware is more complex than for moving the contents of entire (fixed sized) S-buffers. We briefly describe both approaches.

4.1.1 Fixed sized S-cells

One approach is to use fixed sized S-cell images, and allocate enough S-buffer space to accommodate them. Data movement is simplified, because every S-cell is transferred in exactly the same number of shifts.

In cases where data must be moved through a sequence of one or more empty L-cells, an apparent speedup could be obtained by using the shunt-through mode, i.e., passing

data directly from the receivers to the senders of the empty L-cells rather than buffering them in memory.

This reduces the data transfer latency through the empty cells by $M \times N$ shifts, where M is the size of an S-image and N is the number of empty L-cells. But blockage propagation latency is also reduced, as the buffer space of L-cells in shunt-through mode is unavailable to absorb the propagation of blockage. Therefore, it is unclear how much improvement is made to the global data movement cost by using shunt-through mode.

The motivation for using shunt-through mode arises because of the inherently long latency imposed by large fixed sized S-images. Alternatively, the latency could be reduced by using smaller S-images, but the fixed length strategy requires that all S-images be as large as the largest allowable S-cell. This motivates the notion of variable sized S-cells. The additional hardware complexity for handling them is commensurate with that of providing shunt-through as a mode.

4.1.2 Variable sized S-cells

In this approach, the S-buffer sizes are still fixed but the S-images are of varying length, depending on factors such as the states of computation of the various L-cells when storage management commences. It is clearly wasteful to transfer the entire contents of S-buffers from cell to cell. But a design which transfers only the S-images must deal with substantially more complexity.

A simple implementation of variable sized S-cells uses an S-image containing a header of 2-bytes which indicates the size of the image body. The image body is made up of microcode, the values of various registers, and a program counter indicating the state of microcode execution. An empty L-cell contains a null S-cell, i.e., the S-image consists only of a header indicating a zero length body (a word count of two).

4.2 Buffer design

Both the size and usage policy of the S-buffer will strongly affect the blockage propagation properties of an L-cell. Cloning-induced blockage (to be discussed subsequently) motivates larger S-buffers capable of holding multiple S-images. Two usage policies are explored, the more sophisticated one naturally having better performance at greater complexity.

4.2.1 Linear buffering

One advantage of this scheme is that it naturally leaves the S-image located at the beginning of the S-buffer which is convenient for the subsequent computational phases. Another advantage is the simplicity of the hardware implementing it. A major disadvantage of this scheme is that it is prone to excessive propagation of blockage.

Let us describe a simple scenario where blockage occurs. Assume that an L-cell accepts each new S-image by filling its S-buffer from the beginning of the buffer. It is possible to shift an incoming S-image into the buffer which still contains part of an outgoing S-image, provided that the incoming image does not overrun the outgoing one. The transfer protocol must ensure this by suspending shifting-in until there is buffer space available.

Consider the case where a shift-string contains a large S-image followed by a small S-image followed by the rest of the string. The L-cell containing the large image shifts the small image in while shifting the large one out. The transfer of the small S-cell finishes long before transfer of the large one is complete. At this point, no further input is acceptable until the large image is completely shifted out of the buffer. Therefore, the following shift-string is blocked, even though unused buffer space is being created as the large image continues to be shifted out.

4.2.2 Circular buffering

If we allow an S-image to be loaded at an arbitrary location in the buffer, we can improve the blockage characteristics considerably. In particular, if we treat the S-buffer as a circular buffer, we can accommodate any number of S-images in the buffer provided we don't overwrite any used space.

This strategy is not without cost, however. When storage management is complete and the computational phases begin, the S-image addresses must correspond to what the microcode expects. This is not a problem as long as the S-cell is loaded at a fixed address. But in this case, additional storage management overhead is involved to relocate the S-image in its destination L-cell buffer; or additional address translation hardware is needed for the CPU.

We can implement the circular buffer using two address counters (Figure 3), one for receiving and one for sending. These counters increment modulo the buffer size, and roll over to the buffer start address.

Let the values in these counters be *write_address* and *read_address* for the receiver and sender, respectively. A write to the buffer is an atomic operation that stores received data at *write_address*, then increments *write_address*. A read from the buffer is an atomic operation that latches data fetched from *read_address* into the sender, then increments *read_address*.

Let *last_op* be set to *READ* if the last buffer access was a read, or to *WRITE* if the last access was a write. Then the status of the buffer is evaluated in hardware as:

$(read_address\ ==\ write_address\ \&\&\ last_op\ ==\ READ) \rightarrow$ buffer empty
$(read_address\ ==\ write_address\ \&\&\ last_op\ ==\ WRITE) \rightarrow$ buffer full

These conditions are used by the shifter protocol to prevent buffer over-/under-runs.

4.2.2.1 S-cell relocation

Consider the case of relocating the S-cell to the beginning of the S-buffer after data movement is complete. An algorithm is described (see Figure 4) to relocate characters in a circular buffer so that the character at *data_address* is placed at *last_address*, and the characters (circularly) preceding it are moved to their appropriate relocated positions. The following C code fragment implements this algorithm in place. The algorithm takes time proportional to the size of the S-buffer. A more desirable algorithm would be one that takes time proportional to the size of the S-image, while still doing it in place. BUFADDR and BUSIZE are the base address and the size of the S-buffer, respectively, while data_addr is the address of the S-cell header before relocation.

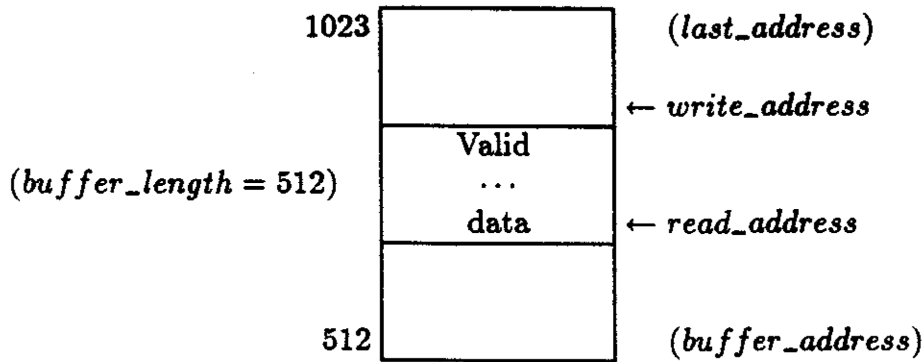

Figure 3: Circular buffer

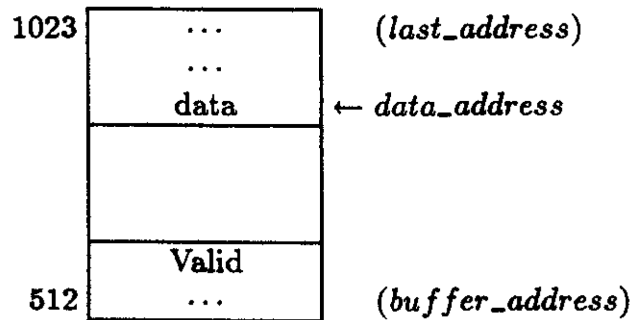

Figure 4: Relocating an S-cell

```
1    delta = (BUFADDR - data_addr + BUFSIZE) % BUFSIZE;
2    curr_addr = old_addr = BUFADDR;
3    val = s_buffer[curr_addr];
4    for (count = 0; count < BUFSIZE; ++count) {
5        curr_addr += delta;
6        if (curr_addr > LASTADDR)
7            curr_addr -= BUFSIZE;
8        if (curr_addr == old_addr) {
9            s_buffer[curr_addr] = val;
10           curr_addr = ++old_addr;
11           if (curr_addr > LASTADDR)
12               curr_addr = old_addr -= BUFSIZE;
13           val = s_buffer[curr_addr];
14           }
15       else {
16           old = s_buffer[curr_addr];
17           s_buffer[curr_addr] = val;
18           val = old;
19           }
20       }
```

4.2.2.2 Address translation in hardware

This section digresses into the computation phases because of the tight coupling of concerns between memory access during computation and circular buffering during memory management.

We may avoid the overhead of relocation by using memory address translation hardware during the computation phases. This hardware should be simple and fast – simple because it must be contained in each L-cell, and fast because it is used for every memory access during computations.

We describe the mod-sum implementation of this function. This implementation is constrained to buffers having $buffer_address = 2^m$ and $buffer_size = 2^n$, where $n \leq m$.

Consider a paged memory having a page size of $2^9 = 512$. Thus, we need a 9-bit address counter to address within a page. The CPU address bus is split into two parts: the lower 9 bits form the intra-page address while the upper significant bits form the page selection address. For all pages except the circular buffer, addressing is straightforward.

If the circular buffer is addressed, however, the 9-bit CPU address is added to a base register containing the negative of the offset of the S-image relative to $buffer_address$. The most significant bit of the sum is discarded, and the remaining 9-bits used to physically address the circular buffer, thus performing the mod-sum operation.

An example is shown in Figure 5. The last word of the S-image is in location 100 of the S-buffer page, and the word count is 313 (acquired from the S-image header during the shifting in process). Thus, the first word of the S-image is at location $(write_address - word_count) \ mod \ buffer_size = (101 - 313) \ mod \ 512 = 300$. Therefore, any CPU address in the S-buffer page is translated according to $(cpu_address + 300) \ mod \ 512$.

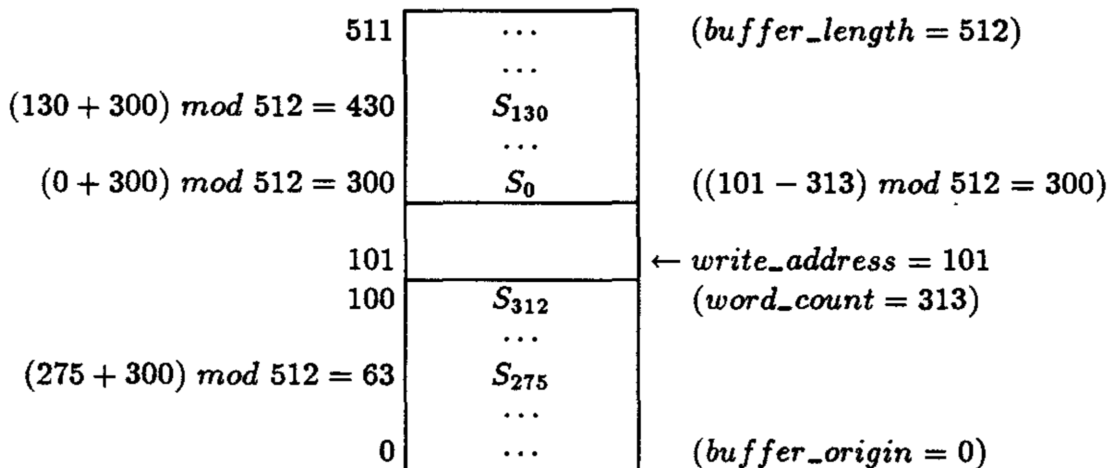

Figure 5: Memory translation

5 Data movement strategies

Various strategies are possible for movement of the S-cells in the L-array during storage management. These will clearly affect the overall performance of the data movement

phase. We have considered several strategies across a spectrum between two philosophical extremes.

A **lazy movement** strategy is never to move data any farther than necessary in any given storage management cycle. This will tend to keep expressions somewhat clumped together, and may occasion cycles where contention for space among adjacent expressions causes very long data movements with correspondingly large performance cost.

A **densely packed** strategy always moves the data such that it ends up densely packed in an arbitrarily located segment of the L-array. While this is rather obviously non-optimal, its main virtue lies in the elegance and simplicity of generating the data movement parameters during storage preparation using Plaisted's algorithm [4].

A **speculative movement** strategy is to spread the FFP symbols out across the entire L-array more or less evenly during each storage management cycle (also implementable using Plaisted's algorithm). This leaves substantial empty space always locally available so that contention between adjacent expressions rarely occurs. But this clearly entails more average data movement, and may therefore exact an unnecessary cost on most cycles.

The performance of any given strategy will depend strongly on the behavior of the expressions under evaluation in the FFPM. Since these are a priori unknown, and themselves dependent on the program mix, it is exceedingly difficult to analyze these strategies for performance in any real application. Running the necessary number of test cases is infeasible under simulation. Therefore, a working hardware prototype is necessary to pursue investigation of this issue.

For the purpose of this report, we assume lazy data movement. An important consequence of this decision is that we never shift an S-cell out of an L-cell that will not eventually be filled by another S-cell during the current data movement phase. This has far reaching effects on how the shifting hardware detects when cloning should occur under the MIRVing strategy, discussed in the next section.

6 Cloning strategies

Under lazy data movement policy, the space allocation mechanism used by the FFPM is exclusively cloning, as follows. During execution, an expression may need more than its currently allocated space to contain its re-written form, and must be suspended until the next memory management cycle. At that time, one of the S-cells of the expression (usually the leftmost one) acquires the additional required space by cloning an appropriate number of copies of itself which are shifted out of the containing L-cell during data movement.

There are a number of ways this can be done. The design choices have profound effects on the creation of blockages and on the complexity of the shifters in the L-cells. The following sections discuss these choices and their ramifications.

6.1 Cloning time

Cloning of a given S-cell could occur at any time during data movement. A uniform strategy must be selected, however, in order to design the hardware which implements cloning. We have considered the following.

The **massed assault**, or pre-cloning strategy launches all the clones from the initial location of a clonable S-cell.

The **MIRVing** (Multiple Independently-targetable Re-entry Vehicle), or post-cloning strategy moves each MIRV (clonable S-cell) to its final destination before cloning it.

The **pregnant S-cell** strategy leaves cloned copies behind as the clonable S-cell is shifted successively to its final destination.

The massed assault strategy, being most obvious, was the first one investigated. But various problems with cloning-induced blockage led us to consider other alternatives. The MIRVing strategy is conceptually as simple as massed assault, and an informal proof has been developed that MIRVing altogether avoids blockage under a lazy movement policy if bidirectional cloning is allowed [5]. The notion of opportunistic cloning of a clonable S-cell at one or more steps en route to its final destination appeared to offer performance advantages, but in general is complex to implement and to understand; the special pregnant S-cell sub-case is functionally equivalent to MIRVing.

6.2 Cloning direction

We assume here that we have adopted a MIRVing policy for cloning. Once the MIRV reaches its final destination and begins cloning, the question arises of which direction its clones should be shifted, which clearly affects what the definition of a ***final destination*** is for a MIRV. Possible strategies are: **left cloning** where all clones are shifted out the left port of any L-cell containing a cloning S-cell; **right cloning** where all clones shift out the right; **unidirectional cloning** where all clones shift out either the left or the right but not both; and **bidirectional cloning** where clones are shifted out both ports.

Unidirectional cloning complicates the data movement control algorithm if flow numbers are used as discussed above. An example illustrates this. Assume a MIRV (clonable S-cell) whose location before data movement is in the middle of a string of L-cells which will be occupied by clones of this S-cell after data movement. Under unidirectional cloning, the MIRV must first be shifted to one end of the string, then clones are shifted in retrograde. Thus, during any given shift cycle, it is not possible for an L-cell to determine from the flow numbers which direction to shift an S-cell. It would be necessary to include a clone count field in each S-image, and the L-cell would have to consider data from this field in all of its currently buffered S-cells along with flow numbers. Since this must all be done in hardware, the cost is prohibitive.

Under bidirectional cloning, the above example is easily handled if we define any such MIRV to be in its final destination before data movement commences. Thus, it begins cloning immediately, shifting clones out both sides in accordance with its left and right flow numbers until they vanish, whereupon the correct number of clones have been shifted out each side to just fill the L-cell string.

7 The transfer protocol

An integral part of the design of the data movement hardware is the design of the packet and transmission protocol. In general, packet transmission protocols are responsible for routing, error control, fragmentation control, etc. In this case, we are concerned only with the efficient and correct transfer of packets from S-buffer to S-buffer, and the detection of the end of the transfer process in each L-cell. Error control is not addressed, and

fragmentation does not occur. Moreover, the network topology guarantees that packets cannot get out of sequence in the transmission process.

Whereas the protocol operates between the sender of one L-cell and the receiver of the next L-cell, we view the process from the point of view of a single L-cell. That is, we describe the protocol as two separate halves communicating with two external entities, and focus on the interaction between these two halves within an L-cell.

7.1 Pure shifting protocol

Consider the case of pure shifting. In this case, there will be at most one receiver and at most one sender active in the L-cell. Both must access the S-buffer. Thus, there are three concurrent processes active in the L-cell during data movement, which must be properly synchronized among themselves as well as between L-cells.

If the left and right shifters each implement sender and receiver functions, the three processes are advantageously modeled as the two shifters and the S-buffer.

7.2 Cloning protocol

In the case of cloning, there are either one or two senders active in the L-cell, along with the S-buffer. Again there are three active concurrent processes. Suitable synchronization is still necessary.

A means for detecting when a MIRV has reached its final destination is necessary to enable the L-cell to switch from a pure shifting protocol to a cloning protocol.

7.3 Algorithmic description of the protocol

A model of these three concurrent (hardware) processes can be written in C-like code if we disregard the details of actual timing. Thus, for example, we do not concern ourselves at this level with whether the S-buffer is implemented as a single- or dual-ported RAM although this clearly will make a significant difference in performance. Also, using C as a medium of expression makes accurate hardware modeling difficult without obscuring the salient features by the details of technically correct code. Discussion and explanation using plain text is interleaved with the C code.

Inter-cell and inter-process signals are modeled as global variables in this code.

```
1    int  left_flow, /* Inter-process communications in L-cell. */
2         right_flow,
3         s_counter;
4
5    struct connection { /* Inter-L-cell communications connections. */
6         FLAG ack, ready;
7         WORD data;} left, right;
8
```

It is necessary to provide some functions to cater to the concurrent nature of the various hardware processes. The following function stubs are intended to fulfill these needs.

```
9    wait(expression) { /* Blocks until expression becomes true. */ }
10
11   int que(function)
12        int function(); {
13        /* Queues calls to function() on a first-come-first-served basis,
14        thus implementing a mutual exclusion mechanism for function();
15        que() returns the return value of function() for each call. */
16        }
17
18   void concurrent(f1, f2, ...)
19        void f1(), f2(), ...; {
20        /* Concurrently executes two or more functions specified by
21        its arguments. Return values of the arguments are ignored.
22        Concurrent() returns when all of its arguments have returned. */
23        }
24
```

When storage management is activated, the current state of the S-cell is contained in the S-buffer. Explicit pointers to the first word of the S-cell, and to one location beyond the last word, are passed as parameters.

The storage management phase begins by invoking storage preparation to calculate the left and right flow numbers for the L-cell.

We presume the S-buffer to be larger than the maximum S-cell size, so equality between `image_start` and `image_end` indicates an empty S-buffer; else there will be exactly one S-cell in the S-buffer; `s_counter` is set accordingly.

The left and right shifters are then concurrently activated. When both shifters have concluded their activities, control is passed to the next FFPM phase.

```
25   void storage_management(image_start, image_end)
26        int image_start, image_end; {
27        storage_preparation(&left_flow, &right_flow);
28        s_counter = (image_start == image_end) ?  0 :  1;
29        concurrent(left_shifter(image_start, image_end),
30             right_shifter(image_start, image_end));
31        exec(next_phase());
32        }
33
```

The buffer process is modeled as reactive, so it is reactivated each time it is accessed. Thus, its state is necessarily modeled as static variables.

During data movement, all calls to `buffer()` are made through `que()` to implement a mutual exclusion mechanism which strictly enforces a first-come-first-served discipline. Thus, at most one instantiation of `buffer()` is active at any time.

Successful calls to `buffer()` will perform reads and writes on the S-buffer; however a read attempted on an empty buffer or a write on a full one will be unsuccessful. Success or failure is indicated by the return value of `buffer()` being 1 or 0, respectively.

A special *peek* operation is used to read a word count without actually *removing* it from the buffer. Otherwise, a read removes a word from the head of the buffer data, and a write appends one to the tail.

```
34  int buffer(buf_op, address, data) /* The S-buffer itself. */
35      int buf_op, *address, *data; {
36      static int write_addr, last_op = READ, s_buffer[BUFSIZE];
37      int buf_stat;
38      if (*address != write_addr)
39          buf_stat = OK;
40      else if (last_op == READ)
41          buf_stat = EMPTY;
42      else
43          buf_stat = FULL;
44      switch (buf_op) {
45          case PEEK:
46              if (buf_stat == EMPTY)
47                  return(0);
48              else {
49                  *data = s_buffer[*address];
50                  return(1);
51                  }
52              break;
53          case READ:
54              if (buf_stat == EMPTY)
55                  return(0);
56              else {
57                  *data = s_buffer[*address];
58                  *address = (*address + 1) % BUFSIZE;
59                  last_op = READ;
60                  return(1);
61                  }
62              break;
63          case WRITE:
64              if (buf_stat == FULL)
65                  return(0);
66              else {
67                  s_buffer[*address] = *data;
68                  write_addr = (*address + 1) % BUFSIZE;
69                  *address = write_addr;
70                  last_op = WRITE;
```

```
71                  return(1);
72                  }
73              break;
74          }
75      }
76
```

The left and right shifters behave identically except for the interpretation of the polarity of their respective flow numbers. Their function is controlled entirely by the flow numbers and the s-counter, which are shared variables. While not explicit in the code, we require that these shared variables are accessed in a mutually exclusive manner (explicit code to enforce this is cumbersome and hinders clarity of presentation). Similarly, we require that the data operations on lines 82 and 83 (similarly, on lines 118 and 119) form an atomic execution unit.

```
77   void left_shifter(data_addr)
78       int data_addr; {
79       int clone_addr = data_addr;
80       int word_count;
81       left.ack = left.ready = 0;
82       while(left_flow-- > 0) { /* acting as a receiver */
83           s_counter++; /* atomic execution with loop test */
84           wait(left.ready);
85           word_count = left.data; /* receive image size */
86           while (word_count-- > 0) {
87               wait(left.ready);
88               while(que(buffer(WRITE, &data_addr, &left.data)));
89               left.ack = 1;
90               wait(!left.ready);
91               left.ack = 0;
92               }
93           }
94       while(left_flow++ < 0) { /* acting as a sender */
95           while(que(buffer(PEEK, &data_addr, &word_count)));
96           while (word_count-- > 0) {
97               while(que(buffer(READ, &data_addr, &left.data)));
98               wait(!left.ack);
99               left.ready = 1;
100              wait(left.ack);
101              left.ready = 0;
102              }
103          if (right_flow < 0 || s_counter > 1) { /* pure shifting */
104              clone_addr = data_addr;
105              s_counter--;
106              }
```

```
107             else { /* cloning */
108                 data_addr = clone_addr;
109                 }
110             }
111         }
112
113 void right_shifter(data_addr)
114         int data_addr; {
115         int clone_addr = data_addr;
116         int word_count;
117         right.ack = right.ready = 0;
118         while(right_flow++ < 0) { /* acting as a receiver */
119             s_counter++; /* atomic execution with loop test */
120             wait(right.ready);
121             word_count = right.data; /* receive image size */
122             while (word_count-- > 0) {
123                 wait(right.ready);
124                 while(que(buffer(WRITE, &data_addr, &right.data)));
125                 right.ack = 1;
126                 wait(!right.ready);
127                 right.ack = 0;
128                 }
129             }
130         while(right_flow-- > 0) { /* acting as a sender */
131             while(que(buffer(PEEK, &data_addr, &word_count)));
132             while (word_count-- > 0) {
133                 while(que(buffer(READ, &data_addr, &right.data)));
134                 wait(!right.ack);
135                 right.ready = 1;
136                 wait(right.ack);
137                 right.ready = 0;
138                 }
139             if (left_flow > 0 || s_counter > 1) { /* pure shifting */
140                 clone_addr = data_addr;
141                 s_counter--;
142                 }
143             else { /* cloning */
144                 data_addr = clone_addr;
145                 }
146             }
147         }
```

8 Hardware design

The algorithm for data movement drives the hardware design. As noted above, the data movement phase requires DMA hardware to shift data at the highest speed achievable within the implementation technology. While the algorithm is capable of correctly handling blockage, it also should run at this maximum speed as long as blockage does not occur. That is, non-blocked shifting should be accomplished in a fully pipelined manner, thus maximizing the utilization of connection bandwidth.

Within a given implementation technology, the maximum shifting rate will be limited by the inter-cell data transfer time, including the accumulated latency of the handshaking, or by buffer access time, including the latency incurred by the buffer contention logic, whichever is longer. An optimal design would balance these two speeds, economic factors being otherwise equal.

The S-buffer can be implemented by a conventional high speed RAM with auxiliary logic to manage access contention, or alternatively with a dual-ported RAM which could double the throughput (assuming the same access time as the conventional RAM), and would incorporate some of the access contention logic internally. In either case, some additional state and logic is required to detect potential buffer over- and under-runs, and to communicate these conditions to the shifter processes.

Each shifter must keep track of its flow number, image word count, current buffer address, and in the case of cloning its clone image start address. It must also respond sensibly to its external interface data and control signals and to the state of the S-buffer.

Whereas our preferred design style is to use Finite State Machine (FSM) methods, it is infeasible to implement all of the required control state in a pure state machine. In particular, counters and registers are implemented as separate entities with necessary auxiliary logic to detect certain significant states, e.g., `(s_counter > 1)` (lines 103 and 139 of the algorithm description). These significant states are treated as inputs to the control FSM(s), while the counters themselves are manipulated by the control FSM(s).

It is possible to implement the communications control structure as three FSMs which directly execute the three processes of the protocol, or alternatively as a single FSM which merges the three conceptual processes into a single physical one. From the design standpoint, a one-to-one correspondence between processes and FSMs is more easily comprehended and therefore desirable. It also generally requires fewer aggregate states to accomplish the required function. Thus, we describe a three FSM implementation of the protocol control structure, i.e., using a buffer controller FSM and two shifter controller FSMs.

Figure 6 presents an abstract block diagram of the hardware. The internal structure of each of the three processes is suppressed, while the explicit state variables are shown. Notice that the three global variables used in the algorithmic description have been embedded into the processes, rather than being openly accessible. Instead, flags representing the significant states of these variables are passed as signals between the processes.

The controllers shown in the functional blocks are FSMs which are responsible for managing all data flow within their respective blocks, as well as synchronization and data transfers between blocks and with the I/O ports.

Figure 6, along with the algorithmic protocol description, is intended to constitute adequate behavioral and high level structural specifications for the design of circuits to

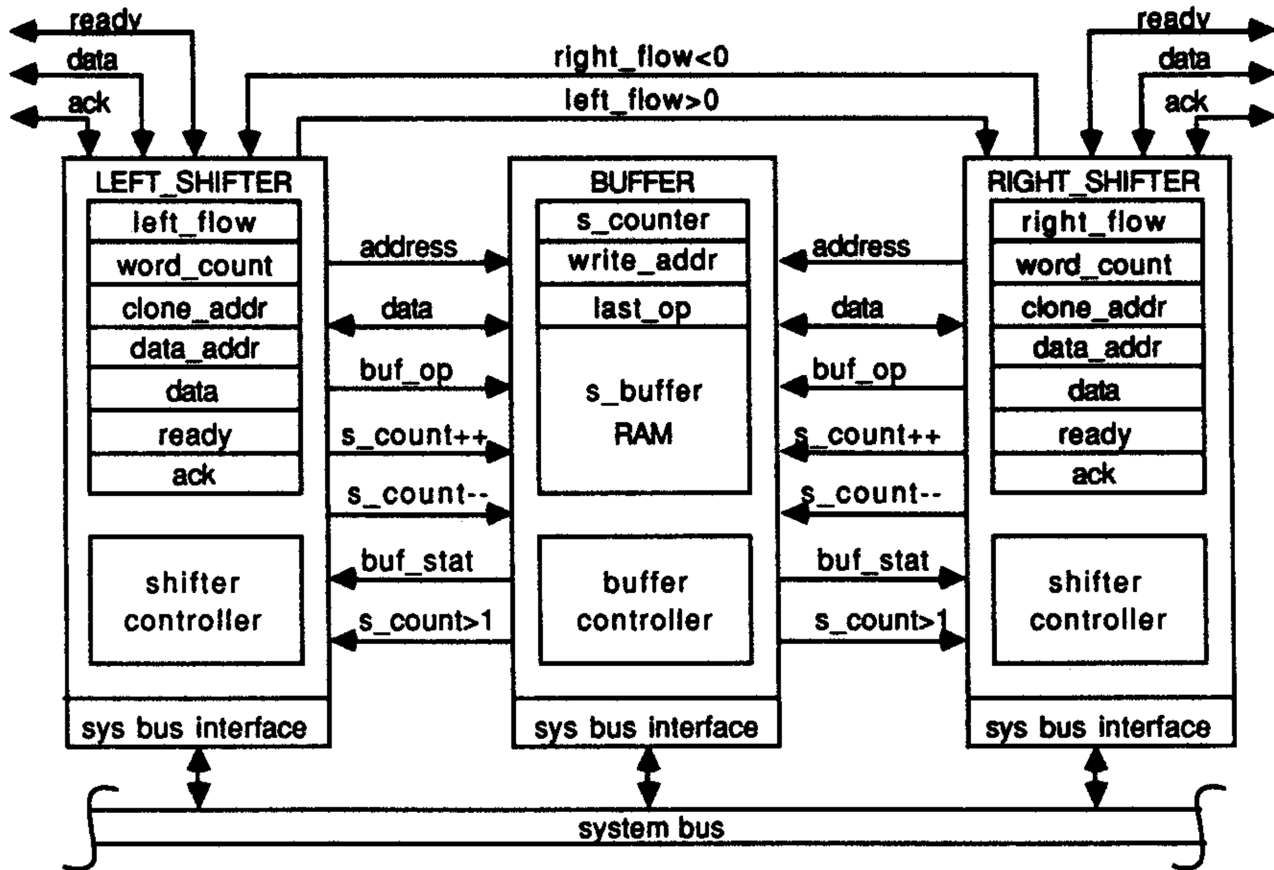

Figure 6: L-cell memory management I/O system block diagram

realize the I/O system. In addition, electrical and timing specifications would be needed, but those are outside the scope of this document.

9 Summary

This report has described the nature of the problem of storage management in the FFPM, and has addressed the major high level design issues. The effects of various strategies were discussed and where possible, comparisons made. The design space is large, with many interdependent decisions. It was possible to explore in depth only a small portion of this space during our study, but a sizable part of it was addressed, at least superficially. Decisions were of necessity often intuitive, with deeper analysis only expended on the most promising alternatives. Consequently, this report follows a more or less single thread of development, noting alternatives without a particularly deep analysis of them.

10 Future directions

The results we report herein were from research directed towards the classical fine grained FFPM. In the course of this research, it was recognized that a coarse-grained implementation of the FFPM architecture offered substantial cost/performance benefits.

Thus, development of the I/O system design should be retargeted accordingly. While many of the results reported here are directly applicable to the coarse grained implementation, a number of additional problems arise. Future research should be devoted primarily to addressing these other problems until such time as a hardware prototype can be constructed. At that time, many questions left open throughout this line of research can and should be resolved.

11 Appendix A: Lazy data movement storage preparation algorithm

Plaisted's algorithm without permuted indices densely packs the data in the L-array. This is neither speculative nor lazy: it clearly moves things around unnecessarily without any speculative benefits. Its main attractiveness is its simplicity and computational elegance.

We borrow the key concept of Plaisted's algorithm, i.e. the conservation of S-cell count, and apply it to a method based on the concept of decompression on demand.

Define:

$$\begin{aligned} C &= F - P \\ &= \text{the compression for an L-segment (segment of the L-array), where} \\ F &= \sum_{i=l}^{r} S_count_i \\ &= \text{the number of cells needed to hold the (Future) expanded expression,} \\ P &= r - l + 1 \\ &= \text{the available space or number of cells in the (Present) L-segment,} \\ S_count_i &= \text{the number of S-cells (original + clones) in L-cell } i, \\ l &= \text{the index of the leftmost L-cell in the L-segment, and} \\ r &= \text{the index of the rightmost L-cell in the L-segment.} \end{aligned}$$

Let L_i, $(0 \leq i < LSIZE)$ represent L-cell i of the L-array. Let N_i be the left flow number of L_i, and by construction, the right flow number of L_{i-1}. Let C_i be the compression of an L-segment containing only a singe cell, L_i.

Initialization at the beginning of storage preparation is performed such that $N_i = 0$, $(0 \leq i < LSIZE+1)$ and $C_i = S_count_i - 1$, $(0 \leq i < LSIZE)$. Thus, if L_i is empty $(C_i == -1)$; if it contains a non-clonable S-cell $(C_i == 0)$; if it contains a clonable S-cell $(C_i > 0)$.

All necessary movement is motivated by regions of $(C > 0)$ (I/O is not explicitly treated here, but input can be faked by $(C_{-1} > 0)$ of a virtual L-cell just off the end of the physical L-array). Thus we treat each L_i having a $(C_i > 0)$ as the kernel of a compressed segment, and attempt to allow decompression according to minimum data movement.

The method is as follows: Each L-cell L_i having a compression $(C_i > 0)$ sends a signal $left_i = (C_i - 1)/2$ to the left and $right_i = (C_i - 1)/2$ to the right. These must be integers, so if C_i is odd, a tie breaker is used to send the extra count to the left or to the right with equal probability, e.g., $\{\, left_i = (odd(i))\ ?\ (C_i-1)/2+1\ :\ (C_i-1)/2;\ right_i = C_i - left_i \}$. The value of C_i is set to 0 when the $left_i$ and $right_i$ are emitted from L_i.

As each $signal_i$ ($left_i$ or $right_i$) propagates away from its L_i, it conditionally interacts with each L_j it passes through, according to $if\ (C_j < 0)\ \{--signal_i;\ ++C_j\}$; and with each N_j it passes by according to $(N_j += right_i \,||\, N_j -= left_i)$. Each $signal_i$ continues to propagate until its value becomes zero or it collides with another $signal_{i'}$.

If $even(i - i')$ the signals collide within an L-cell, say L_k; if $odd(i - i')$ they collide between L-cells, say L_k and L_{k+1}, in which case L_k is arbitrarily selected as being the collision location. In each such L_k, $C_k = (C_k + signal_i + signal_{i'})$.

The process described in the last three paragraphs is iterated (note that the L_k of the previous iteration become the current L_i) until there are no L_k. The worst-case number of iterations is linear in the number of L_i, but the average number is expected to be closer to logarithmic. The lower bound is a single iteration.

12 Appendix B: Modified flow numbers

The flow numbers generated by Plaisted's algorithm conceptually treats each MIRV as $(1 + Clone_count)$ S-cells being moved as a group. A direct implementation in shifter hardware would require extraction of a $Clone_count$ field from each S-cell to be added or subtracted from the L-cell flow numbers during data movement.

An alternative is to treat each MIRV as a single S-cell under pure shifting. This simplifies the shifter's task to performing a unit increment or decrement of the flow numbers, resulting in much simpler hardware. To achieve this, however, it is necessary to modify the numbers generated by Plaisted's algorithm.

Notice that the modified numbers for any MIRV differ only during pure shifting; once the MIRV is in its final destination and cloning, the flow numbers involved reflect the passage of all the clones, and thus are identical with Plaisted flows.

Starting with Plaisted flows, all L-cells containing MIRVs can be found by identifying those having $(Right_flow - Left_flow > 1)$. There are three possible cases: $(Right_flow > 0\ \&\&\ Left_flow > 0)$, $(Right_flow < 0\ \&\&\ Left_flow < 0)$, and $(Right_flow > 0\ \&\&\ Left_flow < 0)$.

In the first case, under a lazy data movement policy the MIRV must shift to the right by exactly its $Left_flow$ to reach its final destination, whence it proceeds to clone. This pre-cloning movement is called pure shifting, and the flow numbers need to be modified exactly over this interval. The second case is similar, but pure shifting will be to the left, by an amount determined by the $Right_flow$. The last case represents one of immediate bi-directional cloning, i.e., the MIRV is in its final destination at the outset, and no modifications to the Plaisted flows are required. In the first two cases, the Plaisted flows are decremented (incremented) by exactly $(Right_flow - Left_flow)$ over the range of pure shifting. A description of this algorithm is shown in C code:

```
1    int modify(flows)
2    int flows[L_SIZE];
3    {
4    int temp[L_SIZE];
5    int i, j, clone_count;;
6
7    for (i = 0; i < L_SIZE+1; ++i)
```

```
8           temp[i] = flows[i];
9       for (i = 0; i < L_SIZE+1; ++i) {
10          if ((clone_count = temp[i+1] - temp[i]) > 0)
11              if (temp[i] > 0 && temp[i+1] > 0)       /* right shifting */
12                  for (j = i+1; j < i+1+temp[i]; ++j)
13                      flows[j] -= clone_count;
14              else if (temp[i] < 0 && temp[i+1] < 0)  /* left shifting */
15                  for (j = i; j > i+temp[i+1]; --j)
16                      flows[j] += clone_count;
17              else if (temp[i] < 0 && temp[i+1] > 0); /* bi-directional */
18              else return(-1);                        /* illegal */
19          }
20      return(0);
21      }
```

13 References

[1] G.A. Magó and D.F. Stanat, The FFP Machine, In *V. Milutinovic (Ed.), High-Level Language Computer Architecture*, Computer Science Press, 1989.

[2] G.A. Magó and R.K. Singh, The Large Grained FFP Machine Architecture, Technical Report, TR89-018, Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[3] G.A. Magó, Storage Management in an FFP Machine, Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, 1989 (in preparation).

[4] D. A. Plaisted, A storage preparation algorithm, private communication, Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[5] V.L. Chi, An Informal Proof of Non-blocking, private communication, Department of Computer Science, University of North Carolina at Chapel Hill, 1989.